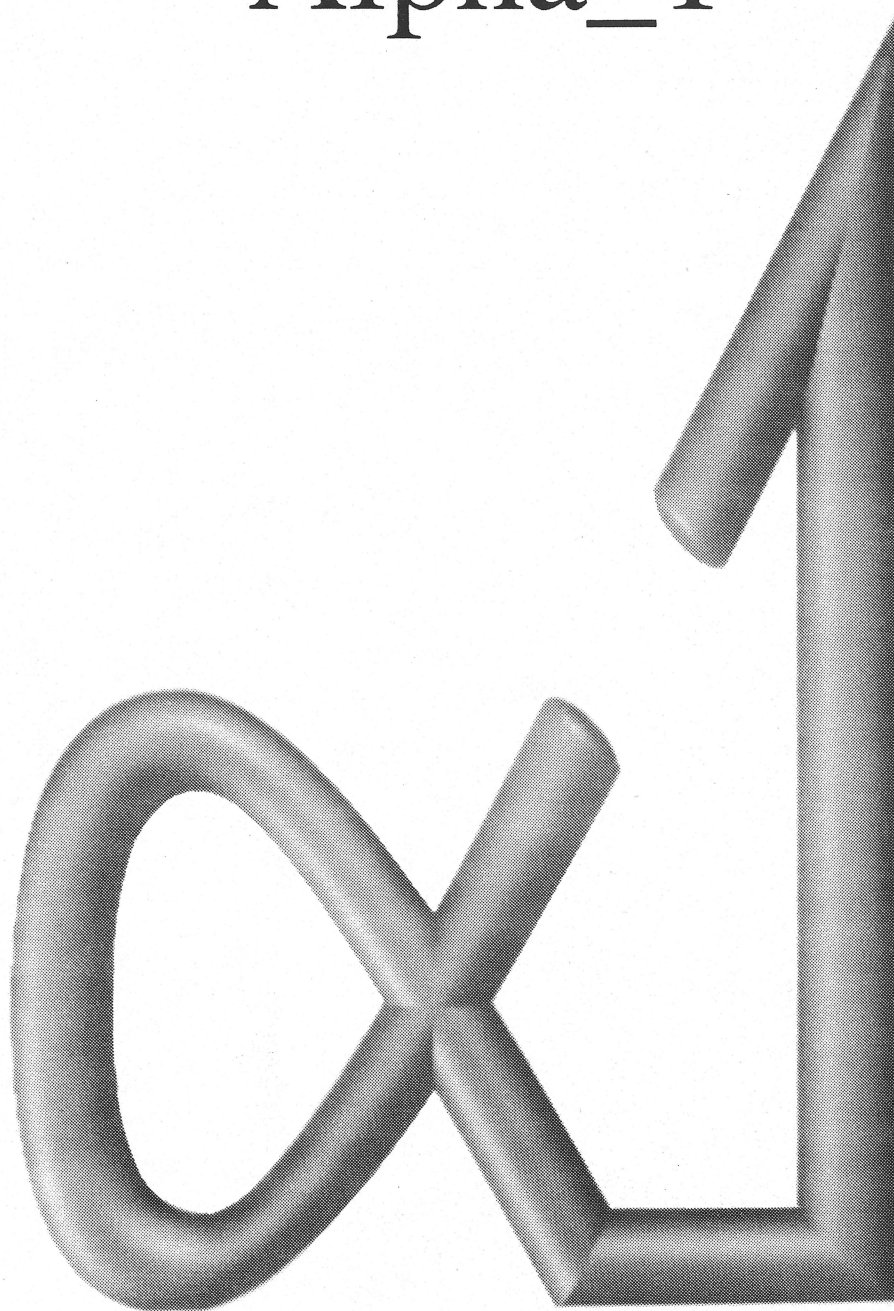


EGS

801 575 6021

born ~~Foster~~

Alpha\_1



User's Manual

# Table of Contents

Overview . . . . .	1
1. Introduction . . . . .	3
2. Environment . . . . .	9
3. Spline Introduction . . . . .	11
4. Shape_edit Introduction . . . . .	19
4.1 General Concepts . . . . .	19
4.2 Programming Interface . . . . .	20
4.3 Graphical User Interface . . . . .	21
4.4 Datatypes in Shape_edit . . . . .	23
5. Orientation Conventions . . . . .	29
6. Graphical Interaction With Shape_edit . . . . .	31
6.1 Screens . . . . .	31
6.2 Windows . . . . .	32
6.3 Displaying Objects . . . . .	35
7. Basic Geometry . . . . .	41
7.1 Sequences . . . . .	41
7.2 Generic Operations . . . . .	43
7.3 Points & Vectors . . . . .	43
7.4 Operations on Points & Vectors . . . . .	47
7.5 Lines . . . . .	51
7.6 Polylines & Polygons . . . . .	57
7.7 Planes . . . . .	58
7.8 Circular Arcs . . . . .	61
7.9 Basic Curves . . . . .	67
7.10 Basic Surfaces . . . . .	75
8. Curve And Surface Geometry . . . . .	83
8.1 Simple Sweeps . . . . .	83
8.2 Volume Primitives . . . . .	85
8.2.1 Basic Primitives . . . . .	85
8.2.2 Rounded Primitives . . . . .	88
8.3 Interpolation & Approximation . . . . .	92
8.4 Surface Construction Operations . . . . .	96
8.4.1 Simple Surfaces . . . . .	96
8.4.2 General Sweeps . . . . .	99
8.5 Unblended Sweeps . . . . .	102
8.6 Blended Sweeps . . . . .	104
8.7 Sweep Examples . . . . .	105
8.8 Derived Surfaces . . . . .	110
8.9 Surface Refinement Operations . . . . .	116
8.10 Surface Modification Operations . . . . .	122
8.10.1 Bending . . . . .	122
8.10.2 Derived From Linear Transformations . . . . .	123
8.10.3 Warping . . . . .	127
8.10.4 Flattening . . . . .	132
8.10.5 Curve-Based Modification . . . . .	135
8.11 Curve Modification Operations . . . . .	137



9. Transforming and Grouping Objects . . . . .	143
9.1 Specifying Transformations . . . . .	143
9.2 Transforming Objects . . . . .	148
9.3 Instancing Objects . . . . .	149
9.4 Grouping Objects . . . . .	150
9.5 Output Optimization for Groups & Instances . . . . .	150
10. Defining New Object Types . . . . .	153
11. Applications . . . . .	157
11.1 Animation Utilities . . . . .	157
11.2 Adding Graphical Dimensions . . . . .	157
11.2.1 Attributes for Dimensions . . . . .	157
11.2.2 Creating Dimensions . . . . .	159
11.2.3 A Dimensioning Example . . . . .	161
11.3 Mechanical Features . . . . .	164
11.4 NC Machining . . . . .	166
11.4.1 Tools for NC . . . . .	166
11.4.2 State for NC . . . . .	169
11.4.3 Generating NC tapes . . . . .	170
11.4.4 NC Tape Block Objects . . . . .	171
11.4.5 Surface Contouring With NC . . . . .	172
11.4.6 Profile NC generation . . . . .	174
11.4.7 Pocketing for NC . . . . .	175
11.4.8 Simulating NC . . . . .	176
11.5 Finite Element Analysis . . . . .	177
12. Leaving Shape_edit . . . . .	181
12.1 Shells . . . . .	181
12.2 Attributes . . . . .	181
12.3 Surface Orientation . . . . .	185
12.4 Saving Data . . . . .	185
13. Input for Utilities . . . . .	189
14. Combining Objects . . . . .	191
14.1 Combiner Overview . . . . .	191
14.2 Preparing a Model for Combining . . . . .	194
14.3 Running the Combiner . . . . .	197
14.4 Combiner Problems . . . . .	199
14.5 Summary of Hints . . . . .	201
15. Rendering Preparation . . . . .	203
15.1 Rendering Guidelines . . . . .	203
15.2 Positioning For Rendering . . . . .	206
15.3 Selecting Shading Parameters . . . . .	207
15.4 Modifying Text Files . . . . .	212
16. Shaded Rendering . . . . .	215
16.1 Running Render . . . . .	215
16.1.1 Render Options . . . . .	216
16.1.2 Scripts For Rendering . . . . .	221
16.2 Shading Parameters . . . . .	222
16.3 Render Flags . . . . .	226

16.4 Render Parameters . . . . .	228
16.5 RLE Utilities . . . . .	228
17. Other Display Options . . . . .	229
17.1 Making Polygons . . . . .	229
17.2 Hidden Line Elimination . . . . .	231
17.3 Fast Polygon Programs . . . . .	233
17.4 Fast Line Rendering . . . . .	234
17.5 Ray Tracing . . . . .	235
17.5.1 Data Preparation . . . . .	235
17.5.2 Invoking Ray . . . . .	237
17.6 Hardcopy With PostScript . . . . .	241
18. Model Analysis . . . . .	245
18.1 Mass Property Calculations . . . . .	245
Appendix A. Summary of Routines . . . . .	249
A.1 Display Control Summary . . . . .	249
A.2 Sequence Summary . . . . .	251
A.3 Points Summary . . . . .	252
A.4 Lines Summary . . . . .	254
A.5 Planes Summary . . . . .	256
A.6 Arcs & Circles Summary . . . . .	257
A.7 Control Mesh Summary . . . . .	259
A.8 Knot Vector Summary . . . . .	259
A.9 Curves Summary . . . . .	260
A.10 Surface Summary . . . . .	261
A.11 Primitive Summary . . . . .	263
A.12 Surface Operators Summary . . . . .	263
A.13 Curve Operators Summary . . . . .	266
A.14 Interpolation Summary . . . . .	267
A.15 Aggregate Summary . . . . .	267
A.16 Transformation Summary . . . . .	267
A.17 Attribute Summary . . . . .	270
A.18 Miscellaneous Summary . . . . .	271
A.19 Applications Summary . . . . .	272
Appendix B. Summary of Menu Entries . . . . .	275
Appendix C. Device Specific Window Managers . . . . .	289
C.1 Silicon Graphics Iris Display . . . . .	289
C.2 X Window System Displays . . . . .	292
C.3 Evans & Sutherland PS300 Display . . . . .	296
C.4 Evans & Sutherland MPS Display . . . . .	297
C.5 Chromatics . . . . .	297
Appendix D. Text File Format . . . . .	299
Appendix E. Rlisp Syntax Summary . . . . .	311
Appendix F. Gemacs & Shape_edit . . . . .	315
Index . . . . .	317





## Overview

The Alpha\_1 User's Manual describes the Alpha\_1 System at a level which is intended to be appropriate for new users of the system. A separate Alpha\_1 Tutorial, included with the User's Manual, may be useful for obtaining an overview of the system without all the detail presented in the User's Manual. The User's Manual is also intended to serve as a reference manual for system developers and application programmers. For more experienced users of Alpha\_1, the function summary in Appendix A and the index will probably be very valuable. The detailed descriptions of low-level functions, internal algorithms, coding standards, and development utilities may be found in the Alpha\_1 System Programmer's Manual.

This manual uses a number of typographic conventions to make it as readable as possible. Technical terms are italicized when they are defined, but thereafter are in the normal text font. The names of functions, programs, global variables, global constants and identifiers are in bold type. The only exception to this is program names that are used repeatedly in a small region of text. In this case, only the first occurrence will be bold face. Text which is to be typed verbatim by users is set in a fixed width "typewriter" font, and text which is to be substituted by users (like a function argument) is in italics. There is a special format which is used for describing functions, and a similar one for programs. The format includes descriptions of all arguments and their datatypes.



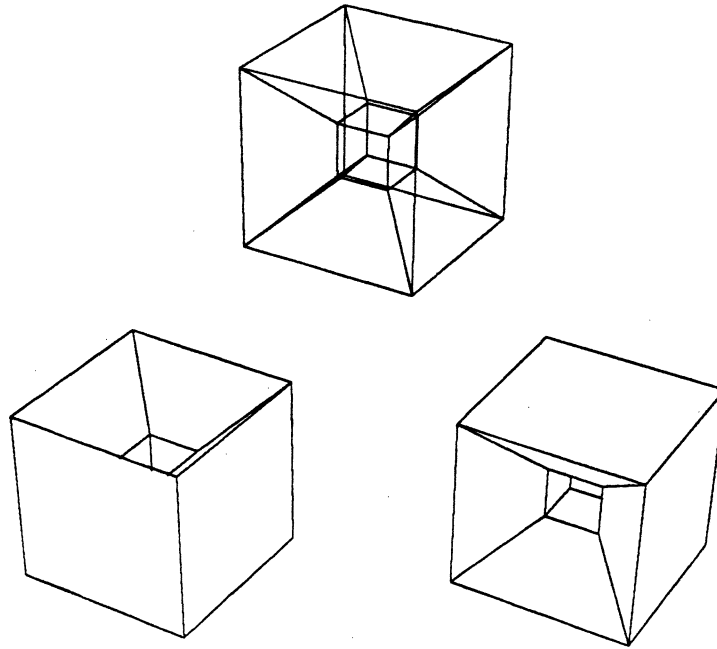


Figure 1-1: An Ambiguous Wireframe Model.

## 1. Introduction

The Alpha\_1 project started in 1980 with the goal of building a spline based solid modeling system. Most commercially available geometric design systems at the time were little more than glorified drafting tables. Some had the capability of drafting in three dimensions, producing "wire frame" drawings. Drawings are, however, inherently ambiguous; Figure 1-1 illustrates this with an example of a wire frame drawing that can be fleshed out into a solid object in several different ways.

### Solid Modeling

At the time, research was well advanced at several locations into modeling solid objects as solid objects, rather than as pictures. Ian Braid and his colleagues at Cambridge University developed the Build system in the late 70's. The Production Automation Project at the University of Rochester, headed by Herbert Voelcker and Ari Requicha had developed the PADL system by this time. Both of these systems modeled solids as a combination, by means of the set operations union, intersection, and difference, of certain primitive shapes, such as boxes, cylinders, spheres, and so on. This style of modeling is commonly called *constructive solid geometry* (CSG), and is able, according to Requicha, to model "95% of conventional, unsculptured parts."

The Build and PADL systems differ, however, in their methods for representing solids. Build uses a *boundary representation*, while PADL uses a *volume representation*. That is, Build stores the boundaries of the modeled solids, while PADL stores a representation of the sets of points which make up the objects. Both representations have their advantages and disadvantages. With a volume representation,

- it is easier to ensure that the model represents a manufacturable object, and



- it is easier to perform set operations.

However,

- the model must typically be kept in an unevaluated form, as primitives combined by set operations,
- it is more difficult to make images of a model, and
- modeling complex shapes is difficult or impossible.

Advantages of a boundary representation are that

- it is easy to make pictures, since the boundary is what is seen,
- the boundary of the result of a set operation may be calculated explicitly, so the model may be kept in an evaluated form, and
- the availability of a wide variety of surface description schemes make it possible to model complex, sculptured objects.

However, boundary representations have disadvantages in that

- calculating the result of a set operation is difficult, and
- ensuring that a boundary really bounds a manufacturable object is difficult.

### Goals of Alpha\_1

The major goal of the Alpha\_1 project was to produce a spline based, boundary representation solid modeler that could be used to model the large class of sculptured mechanical parts which were not representable by a CSG model. Objects which fall into this category include turbine blades, airplanes, helicopters, automobiles, sewing machine parts, shoes, automobile engine blocks, silverware, and so on. Even parts that have, at first glance, a simple geometry sometimes cannot be accurately modeled by a CSG approach. A good example of a part in this class is a bolt. Although it is basically a cylinder surmounted by a hexagonal prism, proper modeling of the threads requires a complex surface shape. Of course, the first representation is probably suitable, except when stress or failure analysis of the bolt is desired.

Alpha\_1 also proposed to demonstrate the utility of high quality graphics in the mechanical design process, particularly when dealing with sculptured surfaces. Proper perception of a complex shape is not possible from a line drawing or rough shaded rendering. Almost all perceived information about the shape of a sculptured object comes from the highlights caused by specular reflections from the surface. Without an accurate lighting model and a high resolution image, the highlights, and thus the perceived shape, will be distorted. Surface ripples that might be invisible in a line drawing show up immediately in a shaded raster image, and errors in design can be corrected before they propagate to the costly machining stage. A good set of pictures of a design can sometimes substitute for expensive scale mock-ups, again saving time and money.

Another goal of the Alpha\_1 project was to incorporate "computer science principles" in the software design. This includes, but is not limited to, use of a modern, structured language, and some adherence to the concepts of data abstraction, object oriented programming, modular design, extensibility, and portability.

Alpha\_1 is also intended to serve as an experimental testbed on which new modeling ideas and techniques can be easily mounted and evaluated. For example, it should be easy to insert new surface representations into the system as they are developed.

### B-Splines in Alpha\_1

The initial surface representation upon which the Alpha\_1 system was built, and the only surface type supported by the entire system, is the rational parametric tensor product B-spline surface.

B-splines have a number of desirable properties for use in computer-aided geometric design. As a spline, a B-spline curve will exhibit a specified degree of continuity; for example, cubic B-splines are continuous in the second derivative. This ensures that a curve which should be smooth is indeed smooth. On the other hand, by properly manipulating the knot vector, discontinuities of specified degree may be inserted in the curve.

B-splines have the *convex hull* property; the curve always lies within the convex hull of the control points. It is therefore easy to bound the curve, simplifying the calculations such as those required for producing an image or for intersecting it with a ray or another curve.

Another feature of B-splines that is useful in a design system is that they provide "local control". Changing the location of a single control point modifies the shape of only a portion of the curve; the rest of the curve remains unchanged. This allows, for example, a designer to shape one region of a curve without affecting other regions that may be already complete.

The use of rational B-splines allows the exact representation of a large class of shapes, including the common quadric surfaces such as spheres, cylinders and cones. A non-rational B-spline, since it is a piecewise parametric polynomial curve or surface, can reproduce a circular or elliptical shape only approximately. By adding a weight or homogeneous component at each control point, it is possible to model these shapes exactly. Since many common mechanical parts have circular cross sections, it is critical that a design system be able to reproduce such shapes.

A B-spline may be evaluated by computing the value at any given point. Often, however, an approximation to the entire curve or surface is desired. This may be found via a "subdivision" or "refinement" process. The Oslo Algorithm provides a method for taking a B-spline and computing a new B-spline, point for point identical to the original, but with more knots, and therefore more control points and more polynomial spans. It can be used to split, or subdivide, a B-spline into two pieces that, together, make up the original. The new control polygon, formed by joining the control points in order, more closely approximates the curve than did the original. Thus, by adding enough new knots, the control polygon itself may be used as an approximation to the curve.

#### A Short Description of the Alpha\_1 System

The Alpha\_1 system provides a number of capabilities. A short listing of the features of Alpha\_1 includes

- an interactive, extensible geometric editor,
- drafting style geometric construction operators,
- high-level shape modification and design operators,
- automatic creation of primitive objects,
- the ability to perform set operations to combine objects into more complex assemblies,
- high quality graphical renderings of object models, and
- computation of simple mass properties of modeled objects.

#### Set Operations

The ability to perform set operations, usually union, intersection, and difference, on models is recognized as one of the characteristics of a true solid modeling system. Set operations perform a number of useful functions in the modeling process. They can be used to combine simpler objects into more complex ones. It is not necessary to design an entire object in one piece; it may be broken down into simpler components that can be designed independently. Thus, set operations provide a facility for modular design. Set operations may be used to model certain common machining operations. For example, set difference models material removal processes such as drilling or milling. Set union may be used to model gluing or welding type operations.

In general, different parts of an object must perform different functions, and may be designed by different techniques. The regions where the different parts meet are not explicitly designed, but arise from the interaction between the independently designed pieces. Using set operations to combine the parts produces the interface region automatically. A good example of this interaction occurs in a turbine blade, where the airfoil portion of the blade must be designed to meet certain aerodynamic or hydrodynamic constraints. The root of the blade is designed to hold the blade into the turbine during operation, and must meet an entirely different set of mechanical constraints. The airfoil must, however, be attached to the root, and a set operation is an ideal way to model the merger of the two.

Performing set operations in a boundary modeling system requires computing the boundary of the result of the set operation. The boundary of the result must be composed of pieces of the boundaries of the operands, and these pieces must be bounded by intersection curves between the boundaries of the operands. Unfortunately, computing the intersection curve between two B-splines in a closed form is currently an intractable problem; the intersection curve between two bicubic surfaces is an implicit polynomial of degree 324. The intersection can, however, be approximated to any desired precision.

The Alpha\_1 system has the ability to perform set operations on "partially bounded sets". As long as they satisfy certain loose criteria, boundaries of objects to be combined by set operations need not be closed. This is a nice feature for the designer, who is not arbitrarily forced to close object boundaries just so that set operations may be performed. A good example of this is seen in the design of a turbine blade root, where none of the boundaries of the subpieces completely enclose a volume, but the final model does have a closed boundary.

### Graphics

A key concept put forth by the Alpha\_1 project is that design of mechanical parts and, in particular, of sculptured mechanical parts, is greatly aided by high quality graphic rendering of the model. The shape of the model can be perceived more accurately and errors in the design found sooner if the designer can better visualize the model being created.

Towards this end, the Alpha\_1 system incorporates a scanline rendering program that can produce images of B-spline surfaces. It adaptively subdivides surfaces until the pieces are flat, to a given resolution, producing polyhedral approximations. These are then rendered traditionally. The program has many options to control the rendering process, the lighting model, surface characteristics, and output form. It is possible to specify transparent or semi-transparent surfaces, allowing the designer to see interior details as well. Control over lighting is also provided, to allow careful placement of highlights to give maximum shape information.

Alpha\_1 also includes a ray tracing program for extremely high quality images. Images created with this program may include shadows, reflections, texture mapping, and other features not available in the scanline rendering program. It will, of course, take much longer to generate this kind of image.

Several support programs are provided to aid in creating good images. One program uses an interactive line drawing display to properly position objects for later rendering. The user moves the image of the object around by manipulating knobs. When the desired view is achieved, the program outputs a transformation matrix that the rendering program will use to create the same view in the shaded image. This capability is also available from the geometric editor.

Another program lets the user manipulate lighting and surface parameters using a specially computed raster image. The position of a light may be varied using knobs or the data tablet, or by indicating the desired position of a highlight. Several lights may be simultaneously positioned, since a single light rarely provides a satisfactory rendering of a complex shape. The reflectance



characteristics of the object surface may also be varied dynamically.

### Geometric Editor

The Alpha\_1 geometric editor, `shape_edit`, is currently implemented in Portable Standard Lisp (PSL). Lisp provides a rich interpretive environment in which arbitrary and extensible data structures may be easily created. The embedding of the editor in a language leads easily to the concept of the program as the model, rather than the output created by the program. It is also easy, therefore, to create a "parametric model", one that can represent a family of parts by changing a set of parameters.

`Shape_edit` provides many tools and operations for shape definition. These include drafting operations, shape modification operations and primitive objects.

The drafting paradigm of design is supported by providing a number of drafting type operations that deal with points, lines, and arcs in the plane. Arcs and line segments may be chained together into a B-spline curve. Curves can be used to construct surfaces in many ways.

To simplify the task of working with B-spline surfaces, high level shape modification and design operators have been created. For example, the "bend" operator takes a previously defined surface and bends it. A thickening operation starts with a single surface, computes a surface that is offset from it by a given amount, and joins the edges of the two to form an object shaped like the original surface.

To accommodate models created by CSG systems, or for those cases when primitive shapes are desired, `shape_edit` contains routines for creating some basic objects. These include boxes, spheres and ellipsoids, cylinders and cones, and tori. The surfaces of the primitives are represented as B-splines, so any of the shape modification operators can be applied to them.

Real objects often have rounded corners, instead of the ideally sharp angles on the primitive shapes. Therefore, provision has been made to create versions of some of the primitives with rounded corners. In particular, cuboidal shapes with rounded edges and corners are supported. The radii need not be identical on all edges, and the faces need not be at right angles. This capability is especially useful when modeling molded parts, since they typically have rounded edges, and must have nonparallel sides so they can be extracted from the mold.

Finally, special purpose routines can be written to model any desired family of shapes. An example of this might be the manufacturing features which can be optionally loaded into the geometric editor. Features such as counterbores, countersinks, and slotted holes have been defined in terms of built-in objects in the geometric editor. Linear and radial patterns of those features can also be constructed.



## 2. Environment

The Alpha\_1 geometric modeling system consists of a set of programs which communicate with each other to model geometric objects, render the objects in various forms for visualization purposes and perform analysis on those objects. Some limited utilities for manufacturing are also available.

The **shape\_edit** program is used for constructing geometric models. **Shape\_edit** is an interactive, interpretive environment for constructing, displaying, and manipulating geometric objects. It is built on Portable Standard Lisp (PSL) using an Algol-like syntax called Rlisp. All of the standard features and programmability of PSL are available within **shape\_edit**, in addition to an extensive set of geometric operations which are loaded into **shape\_edit** for modeling purposes. This results in a very powerful and flexible environment for geometric modeling, although some penalty is paid because the style of interaction is basically a programming paradigm.

An interactive, graphical user interface (GUI) to **shape\_edit** does exist which provides menu-driven access to many of the geometric construction operations included in **shape\_edit**. The graphical user interface provides access to almost all the **shape\_edit** operations, but it sacrifices a few of the powerful control structures which are available from the programming (or command) interface.

For a number of reasons, the major part of this manual is written from the point of view of the programmer's (command driven) interface rather than the graphical user interface. Special aspects of the GUI are described in the appendices, but most of what is described here will carry straight over to the GUI.

A number of utility programs for shaded raster rendering, ray traced rendering, attribute calculations and boolean combination of objects are available to perform additional analysis on parts constructed in **shape\_edit**. These utility programs are written in C, and communicate with **shape\_edit** and each other using either a text or binary form of the data produced in **shape\_edit**.

All of these programs reside in the Unix operating system. Access to the programs is facilitated by an extensive set of aliases and shell variables which provide simple ways to set program flags and invoke groups of programs which work together using single commands. Pipes are used throughout the aliases to channel data from one program to the next so that the programs can be as general purpose as needed. All of the C programs read and write the same binary text format, so the input stream for all the programs is uniform.

An important part of using the Alpha\_1 system for serious modeling efforts in the programming interface or for application development is an emacs text editor. An editor is nice, but not necessary, when working from the graphical user interface. Both gnu emacs and Unipress (Gosling) emacs have been used successfully. A multi-window, screen-oriented editor, emacs has a built-in "info" documentation feature which is used by Alpha\_1 to provide on-line documentation about the system. Access to **shape\_edit** as well as the Unix operating system (the shell) is available from within emacs. Using emacs to access these programs allows simple recording and editing of commands as they are executed by either the shell or **shape\_edit**, since the text is typed into a buffer and then executed. Use of emacs in this manner provides a consistent, uniform environment for reading documentation, editing text, accessing the operating system, invoking utilities written in C, and using the **shape\_edit** interactive modeling program.





### 3. Spline Introduction

This section presents characteristics and properties of B-splines that will be useful in understanding some of the operators described in this manual. The section presents an intuitive introduction to B-splines from a geometric point of view. No rigorous mathematics will be presented because this is readily available in many other places. The goal is to provide enough understanding of the properties and behavior of B-splines so that the operators in the Alpha\_1 system can be used with maximum effectiveness.

Many of the capabilities of the Alpha\_1 system are intuitive and require no understanding of the underlying B-spline representation at all. The reader may initially wish to skip this chapter completely or to skim it lightly, returning for more complete information when it is needed in the context of a particular operation which is to be performed.

#### Curves

A B-spline curve is most simply defined as a piecewise polynomial. Any piecewise polynomial can be represented as a B-spline curve. In computer aided design, we do not generally represent these polynomial pieces explicitly, but use a parametric form instead. The curve may be written as:

$$\alpha(t) = \sum_{i=0}^{n-1} P_i B_{i,k}(t)$$

The curve is completely specified by the following pieces of information:

1. The order (k) of the spline. The order is the greatest degree of any of the polynomial pieces plus 1. A cubic spline has order 4 and degree 3.
2. An ordered set of N control points, the P's, that are simply points in 2D or 3D. This set of N points is called the control polygon for the curve.
3. An ordered set of nondecreasing scalar values (the knots). The set of knots is called the knot vector. A set of basis functions or blending functions (the B's in the equation above) are specified by the knot vector and the order. Each point on the B-spline curve is determined by blending the control points together, weighting them according to the basis functions. A multiple knot is a value that appears more than once in the knot vector. A knot may not have multiplicity greater than the order, and this multiplicity controls the continuity.

None of the control points will in general lie on the spline curve.

Various choices for the configuration of the knot vector result in a variety of end conditions which determine how the spline curve behaves at the ends. A commonly used end condition is the open end condition. A knot vector for a curve with open end conditions will have as its first and last values knots with multiplicity equal to the order. Further, there will be exactly  $N + k$  knots in the knot vector. This means that the curve will always begin at the first control point and end at the last control point. The curve will also be tangent to the line formed by the first (or last) two control points at its beginning (or end).

Alpha\_1 supports two other types of end conditions. With the floating end condition, the knots at the ends of the knot vector have no special configuration. There are still  $N + k$  knots, but multiple knots are not necessary at the ends. A B-spline curve with this type of end condition does not in general touch the end control points at the ends of the curve. Rather, the curve "floats" in the area of the control polygon.

Periodic end conditions are used to create closed curves. There are  $N + 1$  knots in a periodic knot vector and the knot vector is "wrapped around" by equating the first and last knots. The control polygon is the closed polygon formed by connecting the last control point back to the first. Figure 3-1 shows three curves using the same control points, but with knot vectors which produce open, floating, and periodic end conditions.

Although the control points do not in general lie on the curve, it is the case that the curve mimics the shape of the control polygon in a quite predictable fashion. It is this characteristic, more than any other, which has led to the use of B-splines in design. This behavior is due to the variation diminishing property of the B-spline approximation, the convex hull property, and the fact that the B-spline approximation is a local scheme. Details of these important properties are beyond the scope of this discussion, but are easily available in standard texts on B-splines.

Figure 3-2(a) shows a cubic B-spline curve in 2D. The control polygon and the knot vector are also shown. This curve has open end conditions. Note the knots of multiplicity four at either end of the knot vector, and the tangency of the curve to the control polygon at both ends. The open end condition is useful in design because it provides a good handle on where the curve lies at certain critical points. It is also useful for surfaces, because the four corner points of the control mesh are interpolated by the surface. Figure 3-2(b) shows a similar B-spline curve. The only difference in the information that defines these two curves is that one of the control points in the second figure has been moved. This changes the shape of the curve in a predictable manner: the curve is pulled in the direction that the point moved.

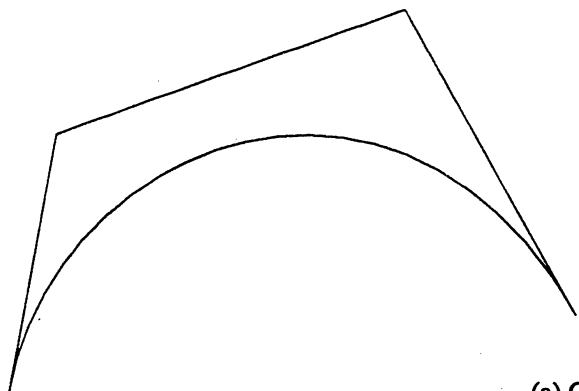
Figure 3-2(c) shows the same control points as in Figure 3-2(a), but this time with a different knot vector. The knot vector also affects the shape of the curve, but not in as predictable a way as the control points. Changing the knot vector affects the way the control points are blended together, and so the curve changes.

Figure 3-2(d) shows another variant of Figure 3-2(a). This time the control points are left the same, and the order of the curve is changed, with appropriate changes to the knot vector. This is a quadratic curve that is tangent to the sides of the control polygon on each edge.

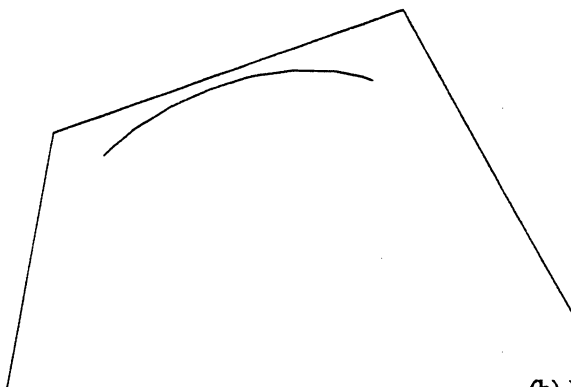
These splines are parametric curves, meaning that the curve is represented as a map from an interval into euclidean 3-space. For the open end condition, the first knot value in the knot vector is the value of  $t$  at the beginning of the curve, and the last knot value is the value of  $t$  at the end of the curve. The curve changes smoothly with  $t$ . It turns out that each knot value corresponds parametrically to a place on the curve where two of the polynomial pieces are joined together, and the knot multiplicity controls how smooth the transition is between the pieces. In a cubic curve with no multiple knots (except at the ends), the polynomial pieces have second degree continuity at the knot values (i.e., two derivatives and the position of the curve match as you approach the point from either side). This is an important property of the B-splines. We could certainly describe the same piecewise polynomial curve in an alternate representation, for example a sequence of Bezier curves. But the B-spline basis guarantees this continuity between the polynomial pieces after adjusting control points, while it must be enforced explicitly for piecewise Bezier curves. This is an extremely useful property for the surface operators, for more on this see chapter 8 [Curve And Surface Geometry], page 83. Surface patches that are "smooth" before a shaping operation is applied will not suddenly exhibit sharp edges afterwards.

The Oslo Algorithm is used to add knots, and hence degrees of freedom, to a B-spline curve without changing the curve. The representation of the curve is changed by adding knots, but not the geometric shape or the parametrization. The operation of adding new knots is called refinement.

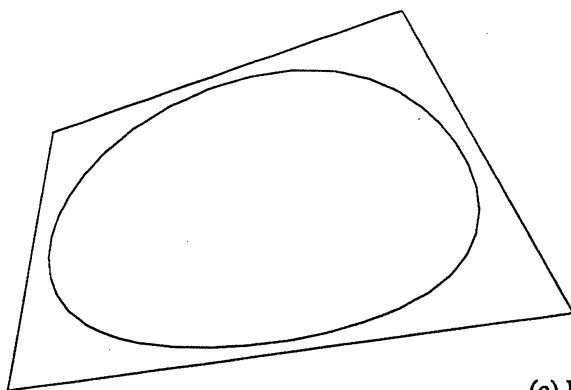
Figure 3-3 shows the result of using the Oslo Algorithm to add different numbers of knots to a cubic curve. Figure 3-3(a) shows the same curve as in Figure 3-2(a), with a single knot added. Notice that adding one knot results in one extra control point being added. This is always the case:



(a) Open End Conditions



(b) Floating End Conditions



(c) Periodic End Conditions

**Figure 3-1:** Different End Conditions

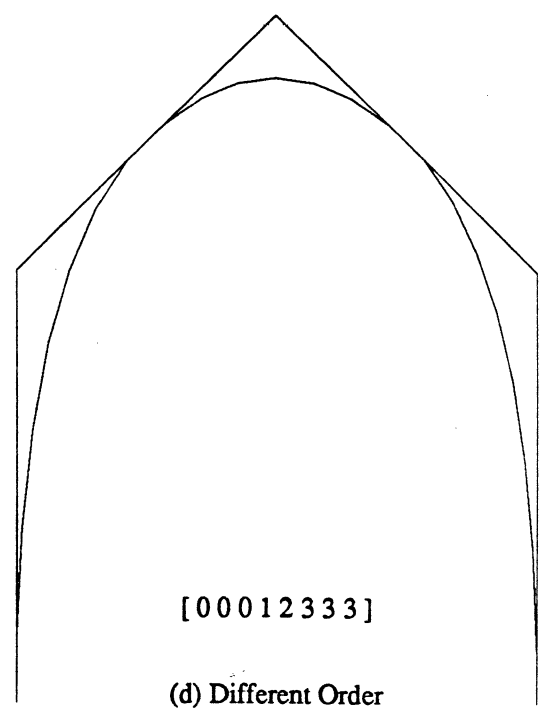
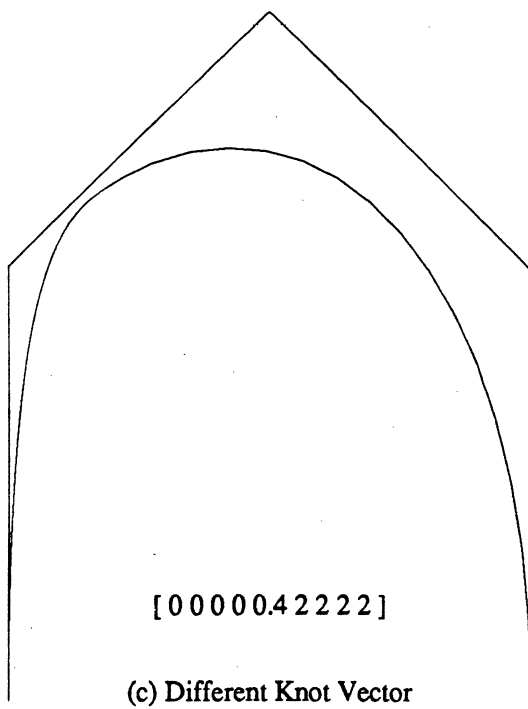
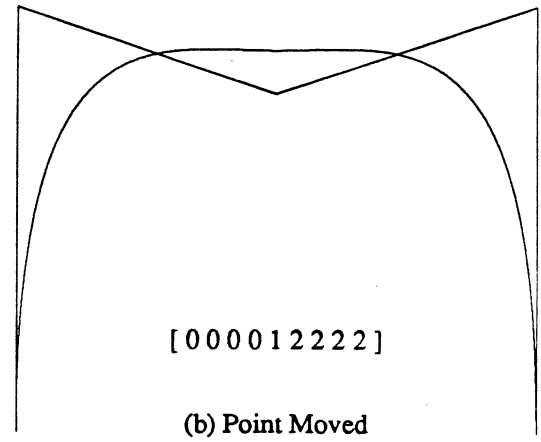
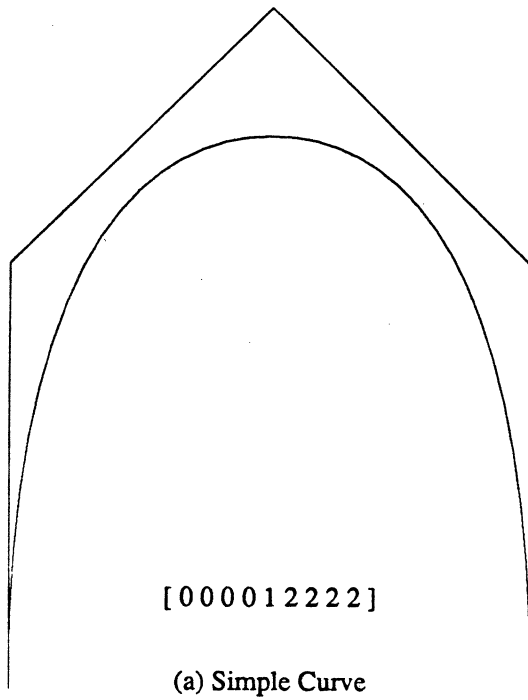


Figure 3-2: B-spline Curves.

addition of  $n$  knots results in the addition of  $n$  control points. The Oslo Algorithm determines exactly where the new set of control points needs to be placed so that the same geometric curve is defined. The addition of knots of multiplicity two, three, and four to the cubic curve are shown in Figure 3-3(b), (c), and (d). The double knot in the first case causes the curve to touch the control polygon, but not at one of the control points. The triple knot makes the curve touch, or interpolate, a control point. This point on the curve corresponds to the parametric value of the triple knot. For curves of other orders than cubic, the same effects can be achieved using knots of appropriate multiplicity. In general, a knot of multiplicity order-1 causes interpolation of a control point as in Figure 3-3(c), while a knot of multiplicity order-2 causes the curve to be tangent along one leg of the polygon as in Figure 3-3(b).

The difference between the triple knot in Figure 3-3(c) and the quadruple knot in Figure 3-3(d) is not visible in the picture. The control point where the curve touches appears twice in the control polygon when a quadruple knot is specified. All the points to the left of the double point pair (including one of the double points) together with all the knots to the left of the quadruple knot (including the quadruple knot) define an open B-spline curve. Similarly, the points on the right hand side (including one of the double points), together with the rest of the knot vector (including the quadruple knot again) form a second open B-spline curve. These two curves together still describe the original curve. We say that the original curve has been subdivided. In general, a knot of multiplicity equal to the order is necessary to perform a subdivision of the curve.

The multiple knots introduce potential discontinuities in the curve. If we moved the points on either side of the control point touched by the curve in Figure 3-3(c), the curve would develop a tangent discontinuity at the control point. This is sometimes useful, depending on the shape desired.

Sometimes, though, we wish to add more detail in a particular region of the curve. If we only have a few control points, moving one of them will affect a large region of the curve. We can add several knots in one region to get more control points added in that area, so the effect is increased local degrees of freedom. Then we have better control over the shape of the curve in that region. Figure 3-4(a) shows a few knots added in one area of the curve. The shape is the same until some of the points are moved (Figure 3-4(b)).

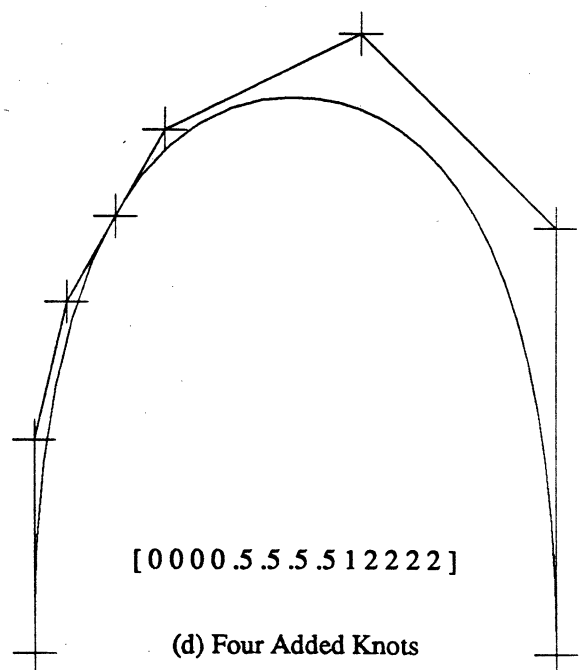
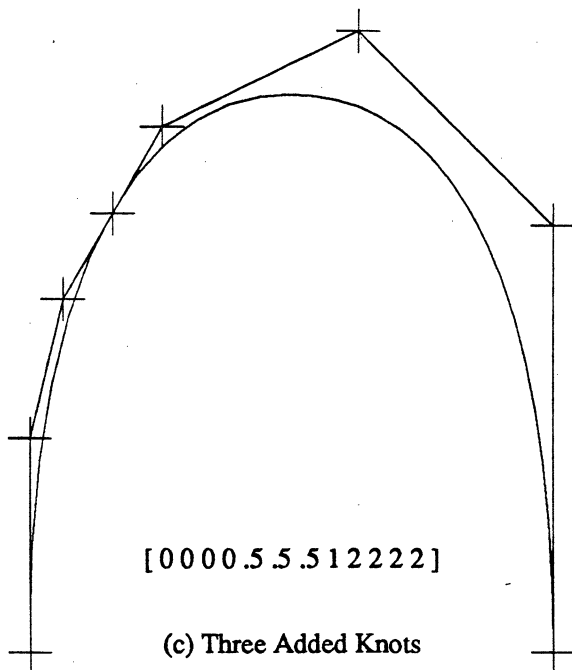
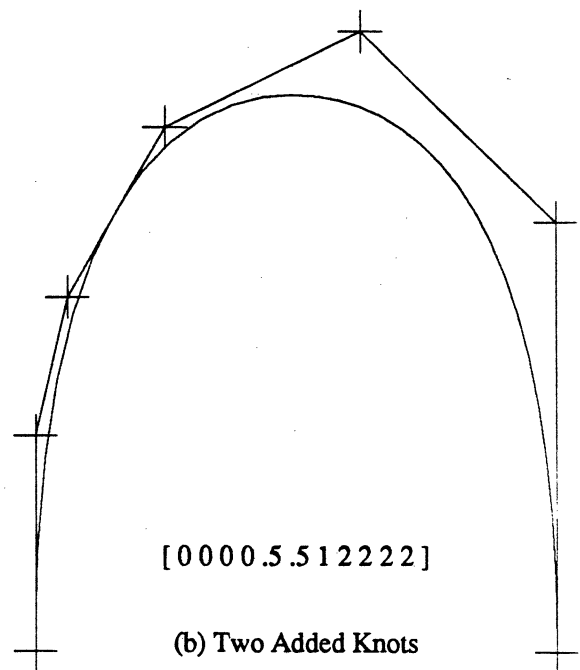
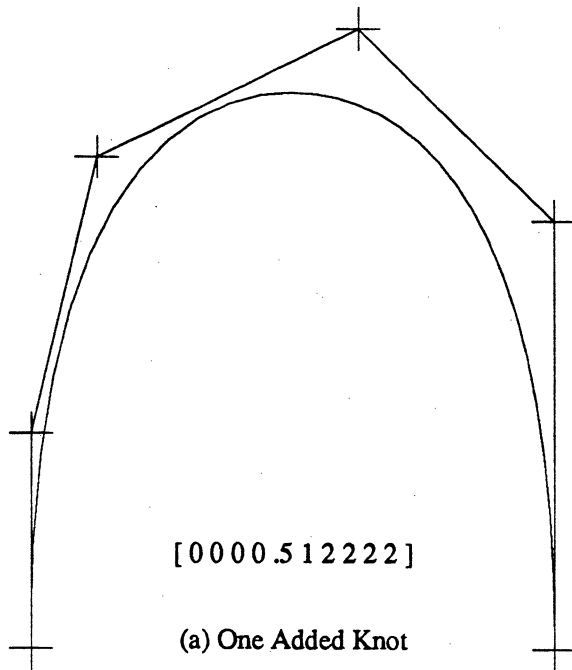
The reader may have noticed throughout this section that as knots were added the resulting control polygon was always "closer" to the curve. The control polygon converges to the curve in that region as the number of knots added is increased. This turns out to be a useful way to draw the curve. Knots are added over the entire parametric range of the curve, and then the refined control polygon is drawn. All the curves in this section were drawn in this way. Enough knots were added to the curve so that the difference between the actual curve and the resulting control polygon was not visible. Though none of the points drawn actually lie on the curve, for equivalent computation cost, the piecewise linear approximation to the curve reflects the shape better than sampling points on the curve and connecting them.

### Surfaces

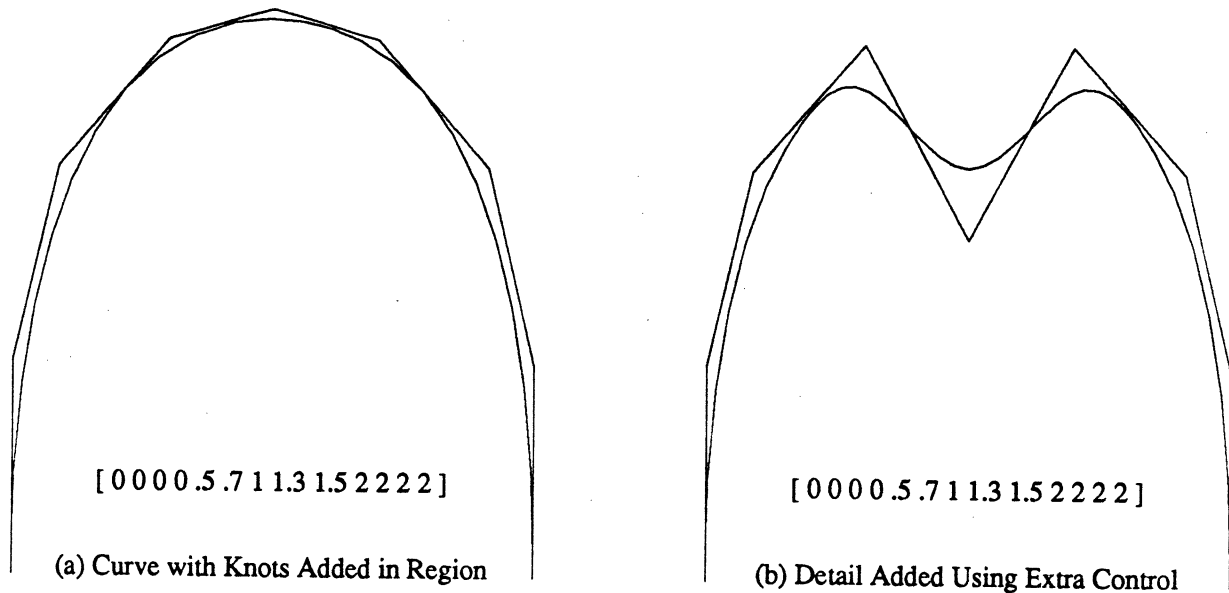
The tensor-product B-spline surface is a simple generalization of the B-spline curves.

$$S(u, v) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} P_{i,j} B_{i,k}(u) B_{j,l}(v)$$

A tensor-product spline parametric surface is a function of two variables which map a rectangle into euclidean 3-space. The surface is typically (but not necessarily) shaped roughly like a rectangle because it is a mapping of a rectangular domain. There are also two orders and two knot vectors, one for each parametric direction. Finally, the control polygon is replaced by a control mesh, which



**Figure 3-3:** Adding Knots to a B-spline Curve.



**Figure 3-4:** Adding Flexibility to a Curve.

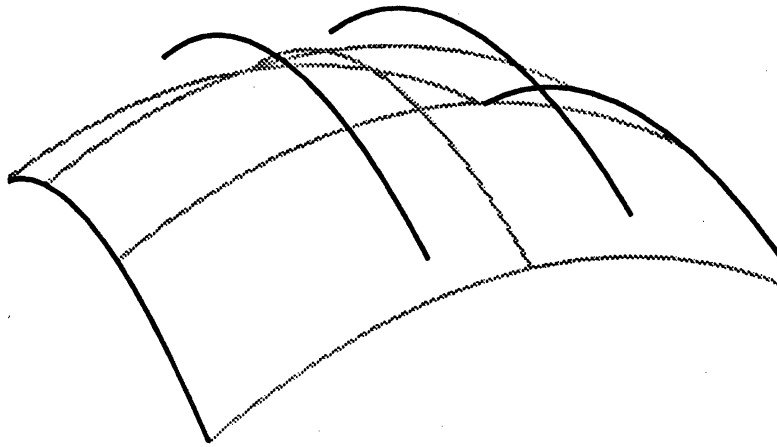
is a rectangular grid of points.

A convenient way to think of the tensor-product B-spline surface is to think of the rows or columns of the control mesh as a set of individual B-spline "control curves" (with the same knot vector and order associated with each of them). The other knot vector and order then describe how these curves will be blended to form a surface, just as we earlier blended points to form a curve. Figure 3-5 shows a surface with the control curves overlaid. These control curves do not in general lie in the surface, but they can be thought of as curves controlling the shape of the surface just as control points determine the shape of a curve.

The surface control mesh behaves the same way as the curve control polygons. If a point is moved, the surface changes nearby. The process of adding knots, though, is slightly more subtle. Notice that there are only two knot vectors, one for each direction, not a separate one for each row and column of the control mesh. This means that if we add a knot to the U knot vector, one control point is added to each row of the control mesh. Changing a knot vector causes the change to be affected all the way across the surface. More degrees of freedom may be added to the surface than are needed, but this is acceptable because the refinement operation changes only the representation, and not the shape or the parametrization of the surface until the extra degrees of freedom are exercised.

Many of the line drawings in this document show surface control meshes. Some of the figures, however, are isoparametric line drawings. An isoparametric line in a surface is simply the curve that is defined by holding one of the surface parameters constant and varying the other. Typically a set of isoparametric lines is drawn to give a better idea of the shape of the surface than either the control mesh or the surface boundaries alone could achieve.





**Figure 3-5:** Control Curves for a Surface.

## 4. Shape\_edit Introduction

The **shape\_edit** program which Alpha\_1 uses for modeling is very different from other existing modelers. This chapter first describes some of the ideas which influence the unique style and approach to modeling that is embodied in **shape\_edit**. The remainder of the chapter introduces the two interfaces (command-driven and menu-driven) which are available in **shape\_edit**, and defines the datatypes for objects which are used in **shape\_edit**.

### 4.1 General Concepts

#### Procedural Modeling

Procedural modeling in the context of the Alpha\_1 system means that the definition of a geometric part is considered to be the procedure or set of commands and parameters which generated it, rather than the set of low-level geometric data such as points, lines, arcs, or spline curves and surfaces which are derived from execution of that procedure. A surface of revolution, for example, may be represented as a set of sampled data points or as a set of spline surfaces. However, neither of these forms is likely to convey the information that the object has rotational symmetry. That information is retained in the procedural description of the part, and may be important for future evaluation, analysis, or documentation.

A related aspect of procedural modeling in the context of Alpha\_1 is parametric modeling. Parametric modeling is a companion of procedural modeling, and simply means that the procedures which define objects are governed by the parameters that they refer too. In more concrete terms, the design of an object may involve definition of a number of named dimensions and other parameters which affect the shape of the object. If one of these parameters or dimensions is changed, the set of commands which perform the construction can be re-executed with no changes to generate a related object with different dimensions. In this way, families of similar parts are naturally supported, as well as allowing parameters to be easily adjusted during the course of the design.

Adjusting a dimension or parameter in the middle of a design is such a common operation that **shape\_edit** provides a mechanism for automatically regenerating all the parts of a model which depend upon a parameter that has changed. "Dependency propagation" records dependencies as commands are executed during an interactive session. Then, if a basic dimension or parameter is changed, **shape\_edit** automatically checks for commands that depend on that dimension and re-executes them, redisplaying the results if necessary.

#### Display as a Side-effect

The display of geometric objects in **shape\_edit** was designed to be as natural as possible, and so geometric objects will generally be displayed as a side-effect of their construction. **Shape\_edit** maintains a notion of a current device which is being used, and the current window in which objects are displayed. So the construction of a geometric object will result in its appearance on the display device. Explicit commands may be given for deleting or adding geometric objects to the display, but these commands also use the notion of the currently active device and window so the designer need not specify those parameters.

#### Display Device Independence

A **shape\_edit** design session is never fixed to use any particular display device, but may use any of the devices which are available. During the design session, the user may request the use of any of the available devices for display. Should he decide during the session that another device has needed capabilities, he can request the use of that device and display objects on that device as well. If no longer needed, he can release the device for other users.

### Groups of Integrated Modeling Tools

The modeling tools available for the design of geometric objects are as unique as the environment and basic approach of **shape\_edit**. Many traditional design tools are available:

- 2D drafting style operations,
- extrusion and revolution operations,
- primitives (cones, cylinders, etc.),
- interpolation and approximation operations.

Some new design tools are also available:

- rounded primitives (boxes with arbitrary radii along the edges)
- high-level shape operators (bend, warp, offset)
- boolean combination operations (union, intersection, difference).

All of these operations are provided in a unified way so that they interact to form a powerful set of design capabilities. Two-dimensional drafting style constructions can be extruded to form surfaces which can be bent or warped. Offsets of rounded primitives can be computed. The boolean combination operations can be applied to surfaces produced from any of the other operations, although the output of the boolean combinations is a final form and cannot currently be used to perform other modeling operations. Of course, analysis and graphical rendering programs operate on the boolean results.

## 4.2 Programming Interface

There are two modes of interaction with **shape\_edit**. One is a command-driven form which we call the *programming interface*. The other is a menu-driven form which we call the *graphical user interface* or *GUI*. The GUI is new, and perhaps less complete than the programming interface, but is much easier for new users to work with. Although some familiarity with computers is helpful in using either interface, one does **not** have to be a programmer to use the command-driven interface. A complete programming language (Lisp) is available in that interface, and so it is a comfortable environment for those who are programmers. But non-programmers need only learn a very small subset of Lisp structures to use that interface. The next section gives a brief summary of the essential syntax for getting started.

For historical reasons, and because it is easier to describe in text form, the main body of this User's Manual is written relative to the programming interface. The functions in the graphical user interface are merely alternate paths to the same functions that are provided in the programming interface. Appendix B of this manual lists all the menu functions for the GUI, describes them briefly, and provides references to the corresponding command interface names where relevant. These names will lead you to the detailed documentation for each function in this manual.

### Rlisp Syntax

The Lisp system upon which **shape\_edit** is built includes a parser that allows a more standard programming language syntax to be used. Only the briefest description of the syntax is given here; it is hoped that the examples in the manual will allow beginning users to pick up the syntax from context. For a more detailed syntax description, or more advanced constructs, see [Rlisp Syntax Summary], page 311.

In Rlisp (the parser used by **shape\_edit**), statements are separated by the “;” character. (Note that it is a separator, not a statement terminator, so it is only necessary between two statements.) Assignment statements use the “:=” symbol, but in **shape\_edit** the result of an assignment will be

displayed automatically if “^=” is used instead. Function calls have their arguments in comma-separated lists enclosed in parentheses, as in most languages. For function calls which have exactly one argument, the parentheses may be omitted (but don’t forget to include them for functions that have no arguments). Indexing is via square brackets (“[“ and “]”) as in C, with a pair of brackets required for each index. Comments are begun with a “%” character and continue to the end of the line.

```
A := 4.0;           % Simple assignment.
B := list( A, A, A ); % Function call.
C := list A;        % Function call with one argument.
D := B[0];          % Indexing.
E := list( B, B );  % A list of lists.
F := B[1][0];       % Two-level index.
```

### 4.3 Graphical User Interface

The graphical user interface to `shape_edit` (the GUI), provides an alternate mechanism for constructing geometric models using the same functions that are available in the programming interface. In the GUI, the major modes of interaction are selecting items from menus, defining points with a mouse or tablet, and picking objects which appear on the display. Currently the GUI is available only for display devices running the X window system, and on the Silicon Graphics Iris display. Device-specific information about using both displays is given in the appropriate section of Appendix C. This section provides a brief overview of the approach and basic features which are held in common across display devices.

Just as in the programming interface, the basic mode for creating a model is executing certain operations which build the geometric entities up piece by piece. The choice of which operation to execute at any time is controlled by selections from menus. Each operation has a certain set of arguments or parameters which must be specified before the operation can be executed. So, building a model in the GUI consists of executing a sequence of operations, supplying the arguments for each operation as they are requested.

This function/argument style of interaction is very simple, but could be very tedious and frustrating if implemented in a rigid manner. The GUI attempts to avoid this by allowing as much freedom as possible within the basic framework. Most arguments can be specified in a variety of ways. Each argument has an associated “type”, corresponding to one (or a set of) of the datatypes described at the end of this chapter. The GUI insists that the type of argument supplied by the user agrees with the type appropriate for each operation. For example, the operation which creates a line through two points must be supplied with two values which are points. But there may be several reasonable and natural ways to specify an object of that type. A geometric entity, like a line or a surface, could be specified by “picking” one that is already displayed. Another way to specify such an entity might be selecting a menu item that constructs that kind of object, and supplying the arguments for that construction. Points are a special geometric entity which can also be specified by “locating” a point on the display screen. Numeric values can be specified by computing them using functions on the “Numbers” menu, selecting predefined constants from the “Numbers” menu, typing them in from a keyboard, or using a graphical “slider bar” to select a value.

In addition to allowing a great deal of freedom in how the arguments are specified, the GUI is not too picky about when they are specified. Although the GUI usually prompts for the arguments in a specific sequential order, it is quite simple to skip around, specifying them in any order which is convenient. As soon as enough arguments have been supplied, the GUI computes the result and displays it, but does not consider the result final until the user “accepts” it. So even when all the

arguments are present, you can use the same mechanism to skip around and adjust parameters until you are satisfied with the result. Furthermore, if you decide to change the parameters after accepting the object, there is a menu item for modifying the object which puts you right back into that same editing mechanism.

### Mouse Buttons

In the GUI on any display, menus are selected using the right mouse button. Usually, the menus pop up at the current mouse position. There are only two levels of menus. Each item on the main menu simply leads to another menu, and each item on a sub-menu is an operation which can be executed. The exact operation of the menus (e.g., whether you let the button up to get to the sub-menus) depends on the individual display.

The middle mouse button is reserved for "pick" and "locate" functions in the GUI. Locators are indicated by holding down the "shift" key while clicking the middle mouse button. A locator simply causes a point to be created at the position indicated by the mouse. Picking (using the middle button without holding the "shift" key) is used to select an item which already appears on the display. The mouse can generally be positioned on any part of the object (although there are a few exceptions on a device-specific basis). Even though the GUI "conditions" the picking so that only objects of the type currently being requested are considered, there are situations where the correct object to pick can't be determined completely automatically, and an essentially random choice is made. For example, if you have created an arc using a point which lies on the arc, it may be hard to select the point because it is on the arc. If you are trying to use an operation that accepts either arcs or points (like **profile**), the system may always choose the arc when you want the point. For these cases, there is a "Pick Types" menu which allows you to specify that you want to select the arc.

The left mouse button is reserved for viewing functions, which vary a great deal, depending on the nature of the display device.

### Keystrokes

The keyboard is used to control actions of the GUI which fall outside the normal sequence of supplying arguments for a sequence of function calls. Before explaining the meaning of the various keystrokes, note that the concept of "nested levels" of functions is implicit in the operation of the GUI. Since any argument could be supplied by selecting another menu item, you could be in the middle of executing several operations all at once. For example, consider constructing an arc tangent to two lines with a given radius. After selecting the menu item, you would be prompted for the first line. If it was on the screen already, you could just pick it, but otherwise you would have to construct it at this point by selecting another menu item. So the construction of the arc is put on hold while you go off and do the line construction. As soon as you finish constructing the first line, the GUI pops right back to the arc construction you were in the middle of, and asks you for the second line.

This nested structure becomes even more apparent when you are editing geometry that you have already created. You can essentially jump back right into the middle of a construction sequence, and move "up" and "down" the various levels to edit other objects that were parameters for the object you selected, or objects for which the selected object was a parameter.

Here is a list of the keystrokes and their actions:

- k** Kill the current function, popping back up one level. If you are several levels deep in a complicated construction, this allows you to change your mind about part of it without starting all over. From the top level, this key means to exit the GUI and return to the command-driven **shape\_edit**

- interface.
- a** Abort everything and start over. This always returns you to the top level of the GUI (but cannot be used to exit the GUI).
  - r** Redo the last command selected from the menu. This is useful if you are using the same function to create several pieces of geometry (e.g., you are making six arcs by specifying two tangent lines and a radius).
  - g, <cr>** Accept the arguments specified for the function and make the resulting geometry part of the model. All operations that have more than one argument wait for you to say "go" before the operation is complete. This is to allow editing of the parameters after display of the results.
  - b** Back one argument. This allows the previous argument to be changed or given initially out of order.
  - f** Forward one argument. Each time you enter an argument, the GUI automatically advances and prompts for the next one. But if you are editing, or giving the arguments out of sequence, you will need to tell it when you want to move forward sometimes.
  - i** Insert another item in a variable length list (before the current item). Many operations allow specification of an arbitrary number of arguments. For example, a polygon is created by stringing together a sequence of points. The single argument to the polygon creation function is the sequence of points, so the points are entered in a sub-level which allows that list to be edited. Normally, the points are entered sequentially, but you can back up and insert another one in the middle with this key.
  - d** Delete an item in a variable length list.
  - h, ?** Help on the current function. This may be useful in getting more information about the operation you are trying to do, but is currently not well tailored for use in the GUI. It uses a very primitive help feature which was designed for the development staff of Alpha\_1.
  - n** Next level down in a construction. This generally comes up when editing existing geometry rather than in creating new geometry. If you are creating an arc from two lines, and the lines have already been specified, you may want to change some of the parameters which were used to construct the line. This key gets you down a level into that parameter list.
  - p** Previous level up in a construction. This is how you get back to where you were.
  - t** Trace the interaction command stack. This shows (in text form) the nested sequence of functions which are currently pending.

## 4.4 Datatypes in Shape\_edit

Throughout the next several chapters which describe **shape\_edit** functions, we will use a set of datatypes to describe the classes of values which are allowed as arguments for functions. These datatypes are listed below with brief descriptions. Some datatypes are specific to certain groups of functions and will not generally be needed. These are indicated where appropriate. The datatypes are given in an order which roughly corresponds to their frequency of use and their order of appearance in the manual.

In function and program descriptions, datatypes are given in angle brackets. If a value may be one

of several datatypes, all the possible datatypes are listed, separated by vertical bars.

`<type1>`

`<type1 | type2 | type3>`

Several of the datatypes are "native" datatypes in the Lisp system on which **shape\_edit** is built.

<b>boolean</b>	These are true and false values. In the Lisp system, these are denoted <b>T</b> and <b>Nil</b> . The <b>Nil</b> value is also a general "nothing" value, serving as a marker saying that no argument is being supplied.
<b>float</b>	A floating point (real) number. Even if a floating point number is required by a function, you can usually use either integer or floating values because the Lisp system converts automatically before doing any arithmetic.
<b>integer</b>	An integer value.
<b>number</b>	<code>&lt;float   integer&gt;</code> If the number datatype is specified in the form <code>&lt;number:v0-v1&gt;</code> the value must lie in the range between <code>v0</code> and <code>v1</code> .
<b>vectorOf</b>	This datatype is always given in conjunction with the type of the elements, as in <code>&lt;vectorOf number&gt;</code> . Occasionally, just <code>&lt;vector&gt;</code> is used to refer to one of these native lisp vector types, without specifying information about the elements. Values of this datatype are usually created with the function <b>vector</b> , which takes a variable number of arguments that are the values for each element. Be careful not to confuse these with the geometric vector datatypes described below.
<b>listOf</b>	This datatype is also always given in conjunction with the type of the elements, as in <code>&lt;listOf number&gt;</code> . Occasionally, just <code>&lt;list&gt;</code> is used to refer to one of these native lisp list types, without specifying information about the elements. Values of this datatype are often created with the function <b>list</b> , which takes a variable number of arguments that are the values for each element.
<b>dottedPairOf</b>	This is an obscure datatype which should occur rarely, but it is prevalent in the native Lisp functions. It can be thought of as a special form of a list with exactly two elements, and is always specified with the types of the two elements, as in <code>&lt;dottedPairOf number,number&gt;</code> . Values which are dotted pairs are created with the "." operator in the lisp system, so <code>(a . b)</code> is an expression which yields a dotted pair.
<b>string</b>	String values are given with double quotes, as in <code>"string"</code> .
<b>keyword</b>	Keyword values are given with just one single quote, as in <code>'LEFT</code> . There is no close quote.
<b>symbol</b>	The <b>shape_edit</b> program defines some special "keywords" which do not require quoting. These are called symbols and are just typed as other words.
<b>function</b>	Occasionally in more advanced operations, the name of a function is given as an argument. Syntactically, these look like keywords: they have one single quote at the beginning.

Two very general datatypes are used in a number of functions. Usually these functions analyze the type of the argument themselves and take the appropriate actions.

<b>object</b>	An object is any of the datatypes that are defined with the objects package. The <b>object!</b> -type procedure returns a non- <b>Nil</b> value for all objects. Most of the datatypes in this list are objects. Exceptions are the native lisp
---------------	---

**anything** datatypes above (numbers, lists, vectors, and keywords for example). Some functions have arguments which actually may take on values of any datatype. The *predicate* functions, like **pointP**, are good examples.

The basic plane geometry datatypes are listed below. The operations for creating and manipulating them are described in the later chapters.

<b>e2Pt</b>	Two-dimensional euclidean points, with X and Y coordinates.
<b>e3Pt</b>	Three-dimensional euclidean points, with X, Y, and Z coordinates.
<b>p2Pt</b>	Two-dimensional projective points, with X, Y and W coordinates.
<b>p3Pt</b>	Three-dimensional projective points, with X, Y, Z and W coordinates.
<b>2DPt</b>	<e2Pt   p2Pt>
<b>3DPt</b>	<e3Pt   p3Pt>
<b>euclidPoint</b>	<e2Pt   e3Pt>
<b>projPoint</b>	<p2Pt   p3Pt>
<b>point</b>	<euclidPoint   projPoint>
<b>r2Vec</b>	Two-dimensional vector, with X and Y coordinates.
<b>r3Vec</b>	Three-dimensional vector, with X, Y, and Z coordinates.
<b>rnVec</b>	N-dimensional vector, with N coordinates, indexed from 0 to N-1.
<b>geomVector</b>	<r2Vec   r3Vec>
<b>anyVector</b>	<geomVector   rnVec>
<b>2DLine</b>	An unbounded line embedded in the Z=0 plane (two-dimensional space).
<b>3DLine</b>	A general unbounded line in three-dimensional space.
<b>line</b>	<2DLine   3DLine>
<b>polyline</b>	A sequence of points. When displayed, they are connected by straight line segments. Polyline with two points are used to represent line segments (since the <line> datatype is unbounded).
<b>polygon</b>	Like <polyline>, but when displayed, the last point first point are also connected by a straight line.
<b>plane</b>	An unbounded plane in three-dimensional space.
<b>arc</b>	A circular arc in three-dimensional space.
<b>circle</b>	A complete circle in three-dimensional space.

The following datatypes are related to spline curves and surfaces.

<b>parmInfo</b>	Parametric information for spline curves and surfaces. It includes polynomial order, end condition types, and knot vector information.
<b>knotVector</b>	The knot vector is part of the parametric information for spline curves and surfaces. It is a non-decreasing sequence of numbers.
<b>knotList</b>	<knotVector   listOf number> Some functions allow knot vectors to be given as a list of numbers, or as a knot vector object.
<b>ctlPoly</b>	The control polygon for curve is usually a sequence of points, although numbers and vectors can also be used in some applications.
<b>ctlMesh</b>	The control mesh for surface is usually a two-dimensional array of points, although numbers and vectors can also be used in some applications.
<b>curve</b>	The spline curve is the only curve representation used in Alpha_1.



**surface** The spline surface is the only surface representation used in Alpha\_1.

A few datatypes for geometric primitives are provided. For display, boolean combinations, and other operations, a spline surface representation is derived.

**box** A box with six rectangular faces.  
**wedge** A wedge with three rectangular and two triangular faces.  
**cylinder** A right circular cylinder.  
**cone** A right circular cone.  
**sphere** A sphere.  
**ellipsoid** An ellipsoid of revolution.  
**torus** A torus.  
**rBox** A very general rounded box which has six faces, although some of them may be "open". Each edge of the box may have an associated radius.  
**rCone** A rounded right circular cone, with associated radii on the top and bottom edges.  
**rCylinder** A rounded right circular cylinder, with associated radii on the top and bottom edges.  
**prim** Any of the primitive objects listed above.  
**paramType** Any primitive or parametric type object (defined with **defParamType**).

Some datatypes for making aggregate structures involving other datatypes are also available.

**shell** Shell objects are special groups of surfaces used by the boolean combination program (**combine**). They contain **only** surfaces.  
**combinerObj** A combiner expression object is a boolean expression describing how a set of shells is to be combined.  
**instance** An instance is a reference to some object which is associated with a transformation matrix. Conceptually, it is a copy of the object to which the transformation has been applied.  
**group** A group is just a convenient way of clustering a number of objects into a single entity for easier reference.  
**matDescr** Matrix descriptor objects describe components of graphical transformations. There are a large number of these, ranging from simple ones like **translateX** to complex ones like **rotateVectorToZAxisWithY**. Transforms (described next) are also considered matrix descriptors.  
**transform** A transform object is a special grouping mechanism for matrix descriptors. Since these objects are matrix descriptors themselves, structures corresponding to standard graphics "transformation trees" can be easily built.  
**matrix4x4** A general 4-by-4 element matrix for cases where the predefined matrix descriptors are insufficient. This datatype should be used with extreme care, and only when absolutely necessary.  
**viewMat** The viewing matrices which are generated by interactive viewing using knobs or slider bars in **shape\_edit**, and retrieved by **getViewMat** are called viewing matrices.

A few miscellaneous geometric datatypes that are used occasionally are:

**attr** Attribute objects are associated with other objects to describe additional

- non-geometric characteristics (like the color of a surface or the width of a line segment).
- bBox** Bounding boxes are the smallest rectangular box aligned with the coordinate axes which entirely encloses the object for which the bounding box was calculated.
- textObj** Text objects are simply strings with an associated position which makes them appear on a display in a particular place.
- diffGeom** When calculated, differential geometry information (like gaussian curvature and principal normals) is stored in a special datatype.

Some of the shaping operations use tables and restriction specifications for some arguments.

- table** <listOf listOf number> This kind of table is used in the **vOffset** operation, and is just a two-dimensional array of numbers.
- restriction** It is sometimes useful to limit the area which a shape operation may modify using restriction specifications. These are used in operations like **flattenR** and **warpR** and are described in detail there.

There are several special datatypes used in the interpolation (curve and surface fitting) package. For the simple curve fitting operations described in this manual, references to these datatypes are hidden from the user level. But they are needed for the more complex general interpolation functions described in the System Manual.

- fitSpec** An object which specifies the information to be used for an interpolation calculation.
- dataItem** A piece of data used in interpolation.
- crvData** A collection of data items to be used for curve interpolation.
- srfData** A collection of data items to be used for surface interpolation.

The animation utilities make use of two other special objects.

- interpVal** Interpolation values are used to "step through" a time sequence, computing new parameters based on fairly arbitrary functions.
- camera** The camera object is used to specify the point of view during an animated sequence.

Finally, a number of objects (knot vectors, control polygons, and control meshes being the most important) are based on a package which packs sequences of floating point numbers into contiguous memory locations. (The native Lisp vectors don't do this.) Advanced users may occasionally need to use functions at this lower level.

- packedVector**  
A packed vector of floating point numbers.
- packedMatrix**  
A packed array of floating point numbers.



## 5. Orientation Conventions

This chapter describes briefly the orientation conventions used in the Alpha\_1 system. All of the conventions are collected here for reference; some will be repeated in appropriate sections of the manual when describing construction of specific geometric types.

All coordinate systems are right-handed. Left-handed coordinate systems are traditional in graphics, but are not appropriate in a modeling system.

All of the following geometric types used by the Alpha\_1 system are oriented: lines, arcs, curves, polygons, polylines, planes, and surfaces. When embedded in a 2D plane, the first four of these can be assigned an "inside" and "outside" based on their orientation:

- The direction vector perpendicular to a line, identifying the halfspace inside the line in the plane, is to the **right** of the line direction. This is consistent with the polygon inside convention.
- The inside of a curve or arc is to the **right** of the curve when traveling along the curve in the direction of increasing parametrization. This is consistent with the polygon and line conventions, and is relevant for orienting a surface produced from a curve by a sweep operation. An alternative statement is that sweep profiles for positive volumes must be constructed with a clockwise orientation in the plane.
- The inside of a polygon is on the **right** when traversing the polygon boundary. This is consistent with the clockwise polygon orientation convention (described below) and the convention that normals point to the inside of a volume (also described below).

The clockwise polygon convention states that when looking at the front (or outside) of a polygon, the order of the vertices is clockwise as shown in Figure 5-1.

Surfaces are somewhat more complicated. The U-V parametrization of a surface is considered to have its origin in the upper-left corner, with U horizontal and V vertical as shown in Figure 5-2.

The control points are indexed (with a zero origin) in [row,column] order. When linearized, the control points appear in row-major order (left to right, then top to bottom).

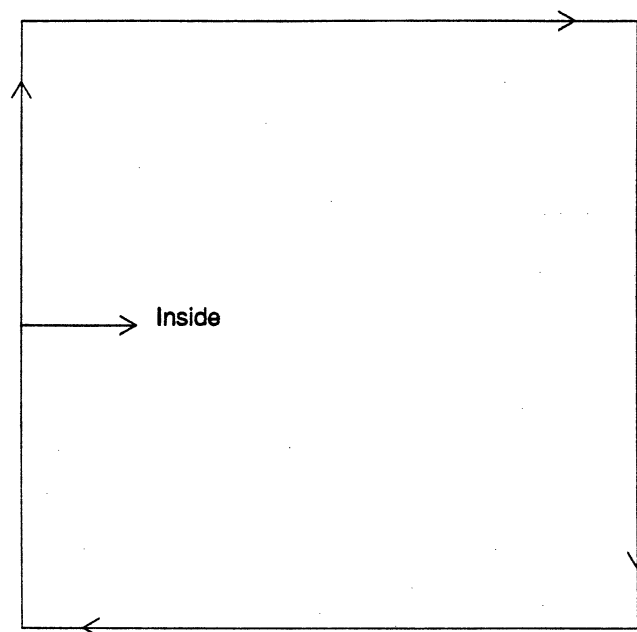
When polygons and surfaces are constructed as the boundaries of a volume, the surface and polygon normals point **into** the volume. That is, they point away from a viewer on the outside.

### Hints for Thinking About Orientation

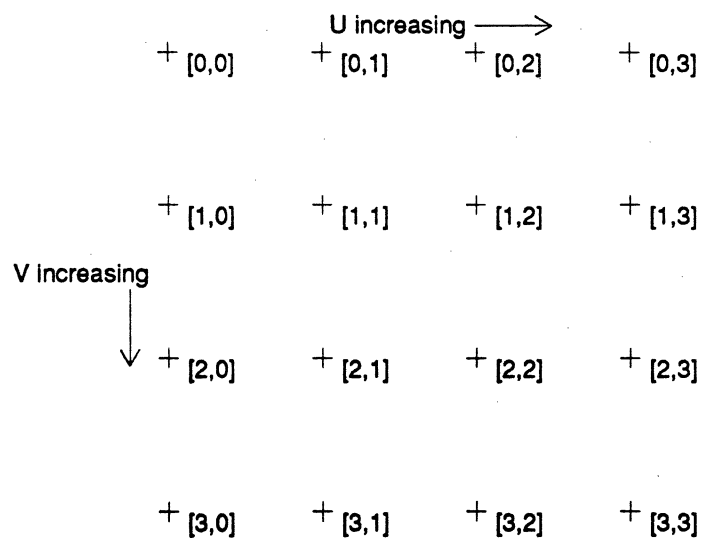
Orientation of objects is very important in Alpha\_1. It is possible not to worry about orientation at all — just reversing the results or adjusting parameters when a computation isn't quite what was expected. However, it will be much easier in the long run to get used to thinking about the orientations of the objects you create.

The most important place to consider orientation is in constructing arcs based on lines. In most other cases, the result looks the same (at least in a line drawing) even if it is oriented backwards. But arcs are constructed by "following" the directions of any lines given as arguments. So if you are building an arc which is tangent to two lines, with a given radius, think about starting on the first line, moving along it in its direction, and then rounding off to follow the direction of the second line as you approach its intersection with the first.

The other crucial place to consider orientation is when constructing geometry for input into boolean combination operations. All surfaces must have their normals pointing to the inside of the volume. Orientation for boolean combinations is discussed further in a later chapter (see section 12.3 [Surface Orientation], page 185).



**Figure 5-1:** Orientation of Polygon, Looking at the Front.



**Figure 5-2:** Orientation of Surface and Control Mesh.

## 6. Graphical Interaction With Shape\_edit

The goal of the display operations described in this chapter is to integrate the management of line-drawing displays with the interactive geometric construction provided by the operations described in the next several chapters. Ideally, the geometry should appear on the screen as a natural side effect of the construction process, with little additional work needed.

An additional goal was to allow access to a large variety of displays using the same mechanism, with a maximum of common underlying code for consistency as well as flexibility to take advantage of the strengths of particular displays.

In the rest of this chapter, components of the display will be described, as well as some commands for controlling the appearance of the objects displayed. A *screen* is associated with the display area of a particular display device. *Windows* are rectangular regions of a screen; there may be many windows on a screen.

A large number of commands are described for controlling the graphics display. Most of the time, only a small subset is needed. The most often used commands are **grab**, **newWindow**, **show**, **unShow**, **removeWindow**, and the “^=” notation, so you may want to pay special attention to those and skim over the rest.

### 6.1 Screens

Typically, when a **shape\_edit** session is underway, there will be a choice of available display devices for graphically presenting the constructions performed. Different displays with different characteristics may be provided and will be chosen depending on the application and on the availability of the particular display.

The **grab** and **drop** commands are used to assign a display device (exclusively) to your **shape\_edit** session, or to release the device. The available devices depend on the local configuration, but the device name might be “PS300”, “Xgen”, “iris”, “mps”, “cx3d”, or others. See [Device Specific Window Managers], page 289 for details of window management on specific devices.

**grab**( *Device* )

*Returns*     <Nil> Select the named device for graphical output.

*Device*     <symbol> The device to select.

**drop**( *Device* )

*Returns*     <Nil> Release a reserved graphics device.

*Device*     <symbol> The name of the device to release.

Grabbing a device that has already been grabbed just makes it the current device. When you drop a device, any windows on that device become inactive and it is no longer the current device. The display will be re-established if the device is grabbed again. Inactive windows are maintained as usual, but windows may not be created on or removed from a dropped device.

If you wish to use the graphical (menu-driven) user interface, you must first use **grab** to assign a display device. Then the **interact** command starts up the graphical interaction. You may switch back and forth between the two interfaces whenever you wish, just by exiting the graphical interaction (using the “k” key) and restarting **interact**.

**interact**()

**Returns** <Nil> Graphical interaction startup, returns when interaction is exited.

Occasionally, you may need to use commands to “flush” all the open devices. *Flushing* does whatever is necessary to bring the display on the device into sync with the current state of the geometric model and internal display lists on the device. It is normally done by `shape_edit`, but will need to be done explicitly in animation drivers that want to show a sequence of display updates while running.

**flush( Device )**

**Returns** <Nil> Complete all pending i/o.

**Device** <symbol> The name of the device to flush.

**flushAll()**

**Returns** <Nil> Complete pending i/o for all devices.

You may remove all windows from a particular device using

**clearDev( Device )**

**Returns** <Nil> Erase any displayed graphics for the device.

**Device** <symbol> The name of the device to erase.

If you wish to switch completely from one device to another, use the `switchTo` command, where the *Device* is the device you want to use from now on. The previous device is dropped, the new one grabbed, and all your windows and graphics copied to the new device.

**switchTo( Device )**

**Returns** <Nil> Move from one graphics device to another.

**Device** <symbol> The name of the device to switch to.

## 6.2 Windows

A screen may have several associated windows. A *window* is a rectangular area of the screen, which has a unique name. It is not necessary to create a window in order to display objects. If the current screen has no window when an attempt is made to display an object, a window named “default” is provided automatically. If you wish to create a window explicitly, use the `newWindow` command, which will create a window with the given name on the current device. If a window with the given name already exists, it is cleared. The `newDevWindow` command allows you to add a window to a device other than the current device. Windows on different devices may have the same name; they are distinct.

**newWindow( Window )**

**Returns** <Nil> Create a window with a particular name.

**Window** <symbol> The name of the window to create.

**newDevWindow( Device, Window )**

**Returns** <Nil> Create a new window on the named device.

**Device** <symbol> The name of the device to create on.

**Window** <symbol> The name of the window to create.

The display manager maintains a *selected window set* of active windows. Any object displayed from `shape_edit` will be displayed in all of the active windows. So it is possible to maintain several views

of a model at the same time. Windows are added to the selected window set as they are created. If the **MultiWindow** flag is off, only one window is active at once, so creating a new window will cause that window to become the selected window set. If the **MultiWindow** flag is on, the new window is added to the selected window set.

on **MultiWindow**

off **MultiWindow**

<boolean> If on, allow multiple active windows (default depends on display device).

A window may be added to, or removed from, the selected window set using the **selectWindow** and **deselectWindow** commands.

**selectWindow**( Window )

Returns <Nil> Add a window to the selected window set.

Window <symbol> The name of the window to add.

**selectDevWindow**( Device, Window )

Returns <Nil> Add a window from a particular device to the window set.

Device <symbol> The name of device to create window on.

Window <symbol> The name of the window to add.

**deselectWindow**( Window )

Returns <Nil> Remove the named window from the selected window set.

Window <symbol> The name of the window to remove.

**deselectDevWindow**( Device, Window )

Returns <Nil> Remove the named window from a particular device.

Device <symbol> The name of the device which has the window.

Window <symbol> The name of the window to remove.

**deselectAll**()

Returns <Nil> Remove all windows from all devices.

Some other possibilities for manipulating windows are copying a window from one device to another. A window is created on the current device with the same name and objects as the designated window. An alternative to copying the window is to move the window to a new device, removing it from the old device. You may also remove a window or clear a window, unshowing all the objects displayed in it.

**copyDevWindow**( Device, Window )

Returns <Nil> Copy the named window from the named device.

Device <symbol> The name of the device to copy from.

Window <symbol> The name of the window to copy.

**moveDevWindow**( Device, Window )

Returns <Nil> Move a window from the named device to the current one.

Device <symbol> The name of the device to move the window from.

Window <symbol> The name of the window to move.

You may remove a window from the current device with



**removeWindow( Window )**

*Returns*     <Nil> Remove the window from the current device.

*Window*     <symbol> The name of the window to remove.

or from a designated device with

**removeDevWindow( Device, Window )**

*Returns*     <Nil> Remove the window from the named device.

*Device*     <symbol> The name of the device to remove the window from.

*Window*     <symbol> The name of the window to remove.

**clearWindow( Window )**

*Returns*     <Nil> Erase any graphics displayed on the named window.

*Window*     <symbol> The name of the window to erase.

**clearDevWindow( Device, Window )**

*Returns*     <Nil> Erase any graphics displayed on the named device and window.

*Device*     <symbol> The name of the device to on which the window appears.

*Window*     <symbol> The name of the window to erase.

The objects which have been displayed in a window may be retrieved (as a "group").

**objectsInWindow( Window )**

*Returns*     <listOf object> Return a list of objects displayed in the named window.

*Window*     <symbol> The name of the window to examine.

**objectsInDevWindow( Device, Window )**

*Returns*     <listOf object> Return a list of the objects displayed in the named window on the device.

*Device*     <symbol> The name of the device on which the window appears.

*Window*     <symbol> The name of the window to examine.

The management of windows varies from device to device, but some things remain constant (if a window manager is provided at all):

- Windows can be repositioned or resized at any time using picking/positioning peripherals. Pushing/popping of windows should be supported on devices with opaque windows.
- Viewing transformations may be modified at any time using valuator (knobs, tablet, slider bars, etc.) There may be a pick action required to initiate viewing modification or to change which window is affected.
- The scale of the view in the window will remain constant on the screen and the "center of attention" will remain in the center of the window as the window is resized. (Scale, translation, and aspect ratio of the viewing transformation will be adjusted to correspond to the new viewport.)

See [Device Specific Window Managers], page 289 for specific windowing commands.

### 6.3 Displaying Objects

The **show** command is used to display an object (or several objects) in the current window(s).

```
show( Obj1, ... )
```

*Returns*     <Nil> Display an object in the current window.

*Obj1, ...*    <Object> The objects to display.

Objects need not be named in order to show them. Thus,

```
show pt( 1, 3 );
```

is a reasonable way to display a point.

In the programmer's interface, a short way of showing an object which you do wish to name is with “**~=**” as in:

```
Point1 ~= pt( 1, 3 );
```

The “**~=**” is a modified form of the “**:=**” assignment statement. It automatically shows the object as well as setting its value.

The **reShow** command is like **show**, but is used when you have modified the value of a displayed object without reconstructing it, so the change does not automatically appear on the display. The **unShow** command erases a displayed object from the current view. **Show** and **unShow** operate on the selected window set, so an object that you show will appear in all of the currently selected windows.

```
reShow( Obj1, ... )
```

*Returns*     <Nil> Redisplay an object in the current window.

*Obj1, ...*    <Object> The objects to redisplay.

```
showInWindow( Window, Obj1, ... )
```

*Returns*     <Nil> Show objects in a given window.

*Window*     <symbol> The window in which to show the objects.

*Obj1*        <object> The object(s) to display.

```
unShow( Obj1, ... )
```

*Returns*     <Nil> Erase an object from the current window.

*Obj1, ...*    <Object> The objects to erase.

```
unShowInWindow( Window, Obj1, ... )
```

*Returns*     <Nil> Unshow objects in a given window.

*Window*     <symbol> The window in which to remove the objects.

*Obj1*        <object> The object(s) to remove.

It is sometimes useful to be able to turn off display during execution of a construction sequence (if a device is unavailable, or if the intermediate constructions are not of interest or are expensive to display). The **Show** switch can be used to prevent any display actions if turned off. If “**off Show;**” has been executed, then the “**~=**” behaves just like the usual “**:=**” assignment operator. If you don't want to use “**~=**”, you can just have all assignments displayed by turning on **showAssignments**. If on, “**:=**” will be treated like “**~=**” and the result of any assignment will be displayed (if it is a displayable object, of course).

```
on Show
```

**off Show**

<boolean> If off, cause “=” assignments to **not** be displayed (default on).

**on ShowAssignments****off ShowAssignments**

<boolean> If no, cause all assignments to be displayed by default (default off).

You may also arrange for any modeling constructions to be displayed, regardless of whether you assigned them a name using the **ShowAll** flag.

**on ShowAll****off ShowAll**

<boolean> If on, display all modeling constructions (default off).

The default values of **ShowAssignments** and **ShowAll** are off. The **Show** flag is off when **shape\_edit** is started, but is turned on when a device is grabbed.

Two other functions, **showPrereqs** and **unshowPrereqs** can be used to display (or erase) a number of objects all at once. The “prerequisites” of an object are all the objects which were used to build it (and the objects which were used to build them, and so on). So if you have made a complicated construction and there is a lot of working geometry on the display, you can get rid of all but the final object using **unshowPrereqs**.

**showPrereqs( Obj )**

*Returns* <Nil> Displays prerequisites of the specified object.

*Obj* <object> The object for which prerequisites are to be displayed.

**unShowPrereqs( Obj )**

*Returns* <Nil> Remove prerequisites of the specified object from the display.

*Obj* <object> The object for which prerequisites are to be removed.

Sometimes highlighting certain objects on the display can be helpful, so several functions are provided for that. Highlighting varies depending on the display. A device with color may change the color of highlighted object, while a monochrome calligraphic display may just draw the object twice so it appears brighter.

**highLight( Obj )**

*Returns* <Nil> Highlights the specified object.

*Obj* <object> The object to be highlighted. It should already be on the display.

**showHigh( Obj )**

*Returns* <Nil> Displays an object and highlights it.

*Obj* <object> The object to be displayed.

**unHighLight( Obj )**

*Returns* <Nil> Display a highlighted object in normal mode.

*Obj* <object> The object for which highlighting is to be turned off.

A few other switches and global variables are used to control the way spline curves and surfaces are displayed. For curves, the default display is to draw a smooth curve, and not draw the spline control polygon. The **CrvPolys** and **SmoothCrvs** switches allow the curve display to be adjusted.

The default settings are Nil for **CrvPolys** and T for **SmoothCrvs**. You can check the current settings using **getCrvPolys** and **getSmoothCrvs**. Turning off both switches will result in nothing appearing on the screen.

**setCrvPolys( Flag )**

*Returns* <Nil> Control display of spline control polygons.

*Flag* <boolean> If true, display control polygons (default false).

**setSmoothCrvs( Flag )**

*Returns* <Nil> Control display of smooth splines.

*Flag* <boolean> If false, do not display smooth splines (default true).

**getCrvPolys()**

*Returns* <boolean> The value of the **CrvPolys** switch.

**getSmoothCrvs()**

*Returns* <boolean> The value of the **SmoothCrvs** switch.

In addition, the "smoothness" of a displayed curve may be controlled with the **CrvFineness** parameter. The default value is 1.0, and the current value can be checked with **getCrvFineness**. This parameter controls the accuracy of the line segments that are used to represent the curve on the display. Smaller values will result in more line segments used to display curves and the computation will be more expensive.

**setCrvFineness( Value )**

*Returns* <Nil> Set the smoothness of displayed splines.

*Value* <number> New smoothness factor, larger numbers make coarser curves (default 1.0).

**getCrvFineness()**

*Returns* <number> The current smoothness factor.

Surfaces also may be displayed as a set of smooth curves, as a control mesh, or both. The **SrfMeshes** and **IsoLines** switches control the type of display. The default is to display the smooth curves only (**SrfMeshes** is Nil and **IsoLines** is T). You can check the current values with the **getSrfMeshes** and **getIsoLines** commands. The number of isoparametric curves which are shown in each direction of the surface is controlled by the **SrfFineness** parameter, which can be set using **setSrfFineness**. The default is 50.0. Use **getSrfFineness** to check the current value. Smaller values will result in more curves being displayed in each surface direction, and the computation will be more expensive. Note that **SrfFineness** controls the spacing of the isoparametric curves; use **CrvFineness** to control their smoothness.

**setSrfMeshes( Flag )**

*Returns* <Nil> Control display of surfaces as sets of control meshes.

*Flag* <boolean> If true, display control meshes for surfaces (default Nil).

**getSrfMeshes()**

*Returns* <boolean> The current setting of the **SrfMeshes** flag.

**setIsoLines( Flag )**

*Returns* <Nil> Control display of surfaces as sets of smooth curves along the surface.

*Flag*           <boolean> If true, display smooth curves in the surface (default T).

**getIsoLines()**

*Returns*       <boolean> The current setting of the **IsoLines** flag.

**setSrfFineness( Value )**

*Returns*       <Nil> Control the number of isolines displayed in each surface direction.

*Value*           <Float> The new setting of the **SrfFineness** parameter, smaller values generate more curves in the surface (default 50.0).

**getSrfFineness()**

*Returns*       <number> The value of the **SrfFineness** parameter.

Surfaces may be displayed with normals at the four corner points as an aid to checking orientation. (Recall that normals should point to the inside of an object). The default value is Nil, for no normal display. You may reverse the direction of the normal display using **setReverseNormals** if needed. Sometimes the normals are difficult to see inside the object on a linedrawing display, and this option will make them more visible. But remember that you are seeing the reverse! You can check the current values of these two flags with **getSrfNormals** and **getReverseNormals**.

**setSrfNormals( Flag )**

*Returns*       <Nil> Control display of surface normals at the corners of surfaces.

*Flag*            <boolean> If true, display surface normals (default false).

**getSrfNormals()**

*Returns*       <boolean> The current setting of the **SrfNormals** flag.

**setReverseNormals( Flag )**

*Returns*       <Nil> Cause displayed surface normals to point in the opposite direction.

*Flag*            <boolean> If true, display normals in opposite direction (default false).

**getReverseNormals()**

*Returns*       <boolean> The current setting of the **ReverseNormals** flag.

For some surfaces with "degenerate" corners, the normal display is not very useful because the normals at the corners cannot be easily computed. It might be necessary to use the **shrinkIt** procedure to derive a slightly smaller part of the surface which does not have degenerate normals at the corners, but which has the same orientation as the initial surface.

**shrinkIt( Srf )**

*Returns*       <surface> Computes a surface which is a little smaller than the original and unlikely to have degenerate corners.

*Srf*             <surface> The surface from which the new version is to be derived.

It is occasionally useful to be able to retrieve the current viewing matrix associated with a window on the display. The **viewMat** function waits for a viewing matrix event to be initiated by interaction with the display (the actual actions depend on the particular display device) and returns a special object representing that event. This object can be sent to other windows to reproduce the same view, or dumped into an Alpha\_1 text file (via **dumpA1File**) for use with the **render** program.

**viewMat()**

**Returns** <viewMatEvent> An object representing the current viewing matrix.

**setWindowViewmat( WindowName, ViewMat )**

**Returns** <Nil> Set the viewing matrix in a particular window.

**WindowName**

<symbol> The name of the window in which to set the viewing.

**ViewMat** <viewMatEvent> The new viewing matrix for the window.

**setDevWindowViewMat( DeviceName, WindowName, ViewMat )**

**Returns** <Nil> Set the viewing matrix in a particular window on a particular device.

**DeviceName**

<symbol> The device where the window resides.

**WindowName**

<symbol> The name of the window in which to set the viewing.

**ViewMat** <viewMatEvent> The new viewing matrix for the window.

Text strings can be displayed using the same display functions as other objects, but you must first create a text object to hold the string and the location at which it will be displayed.

**text( Location, String )**

**Returns** <textObj> A text object.

**Location** <euclidPoint> The location of the string (lower left corner).

**String** <string> The text.

### Dependency Propagation

In order to understand the important concept of *dependency propagation* in **shape\_edit**, we must understand what a *model* is. Every geometric object created in **shape\_edit** (lines, arcs, surfaces, etc.) has extra information stored with the geometric data. This extra information describes how the object was created, what other objects were used to create it (objects it “depends” on) and what other objects used it when they were created (objects that “depend” on it). So any object is actually part of a network of objects, which we refer to as “the model.” This information is usually hidden, but you will probably appreciate the effects of its invisible presence!

By default, **shape\_edit** propagates dependencies between constructed objects. This means that if you create several lines, construct an arc from those lines, and then change one of the lines, the arc will be automatically updated (and redisplayed if necessary). When you are first sketching out a model, or trying various dimensions and parameters, this is an extremely useful feature.

There are, however, some situations in which it is much more useful not to update all objects which depend on every item that you change. In these situations, you may want to turn off the dependency propagation with the **propagateChanges** switch.

**on PropagateChanges**

**off PropagateChanges**

<boolean> If true, modifications are propagated to the rest of the model (default true).

A common situation where turning off the propagation is useful is when you are nearing completion of a model. If you have already executed a large number of constructions in your **shape\_edit** session, and you wish to go back and change something near the beginning of the construction, a lot of computation time can be needed to regenerate the entire succession of objects that depend on each other. (Essentially, your entire model is rebuilt!) If you have made an error, then it may be obvious from just the next several statements that you need to change the object again. So you may want

to turn off **PropagateChanges** if you want to make local changes in something that is part of a large construction.

Another common situation is if you have a sequence of several statements which are fairly expensive computationally, and you re-execute them as a block. In this case, change propagation can cause the construction to be essentially re-executed completely for each statement in the block.

In the programmer's interface, when **propagateChanges** is turned on (and remember that is the default), you need to be careful with your variable names. If you use a variable name for one construction, and then re-use the name in another construction, the first construction will be changed! This usually occurs when you try to write a loop with an assignment statement in the loop. The solution is usually to rewrite the code in another way, rather than to turn off **PropagateChanges**.

A related problem occurs when you use the same variable name in a sequence of statements. The sequence:

```
Crv1 := cRefine( Crv1, ROW, list( 0.5 ), T );  
Crv1 := reflect( Crv1, 'x );
```

works fine. The dependency propagation checks for and correctly handles single level references to the same name within a statement. But:

```
Crv1 := reflect( cRefine( Crv1, ROW, list( 0.5 ) ), 'x );
```

causes an infinite loop, because the reference is through two levels. This code should also be rewritten, rather than just turning off **PropagateChanges**. It is usually preferred that different names be used anyway, since the dependency scheme can't remember two different constructions for the same object.

## 7. Basic Geometry

Alpha\_1 provides a set of basic geometric primitives and operations in 2D which are similar to those used in engineering drafting. Since the Alpha\_1 system is intended for use in modeling sculptured objects in 3D, the geometric constructions in 2D are used primarily as a familiar stepping stone into the higher level operations on curves and surfaces which will be described later. Depending on the model, a relatively small part of the modeling process may involve these 2D constructions compared with more traditional engineering/drafting approaches.

Coordinate geometry is used to represent dimensioned distances. Both linear and angular dimensions are just numbers, and can be freely intermixed integers and real (floating point) numbers. Most parts and sub-parts of models will be designed in a convenient local coordinate system so that the dimensions and distances which govern the design are aligned with a major axis. Then the dimensions can be used directly in the geometric constructions. Angles are by default measured in degrees, counter-clockwise from the right (+X) direction.

### 7.1 Sequences

Sometimes there are tables of related dimensions which must be manipulated as a group. There is a group of commands which fit with the list manipulation capabilities of the PSL language underlying `shape_edit`, constructing lists of numbers which represent sequences of dimensions.

The list primitive takes a variable number of arguments and returns them in a sequence. This is an important construct for grouping a number of objects, and many operations require lists of objects as arguments.

`list( Arg1, ... )`

*Returns*     <listOf anything> Collect the arguments into a list.

*Arg1, ...*     <anything> These are the values which you want to make into a list, they can be any datatype known to Alpha\_1.

An item may be repeated *N* times, constructing a sequence of the same value. A sequence of values "ramped" (linearly interpolated) between two values can be constructed with `numRamp`. Using `numStep` yields a sequence of values with a given starting value and stepsize. Several sequences may be combined into a larger sequence with `append`. In combination with the "foreach...collect" loop, the integer sequences from `seq1` and `seq0` can be used as a basis for further sequences. The former returns a sequence of integers from 1 to *N*; the latter a sequence from 0 to *N*.

`nTimes( N, Item )`

*Returns*     <listOf anything> A list of *N* Items.

*N*             <integer> The number of times to repeat the item.

*Item*          <anything> The item to repeat. Note that *Item* is evaluated only once.

`numRamp( N, Num1, Num2 )`

*Returns*     <listOf number> A list of *N* numbers between *Num1* and *Num2* inclusive.

*N*             <integer> The number of values to calculate for the list.

*Num1, Num2*

             <number> The starting and ending values of the list, respectively.

`numStep( N, StartVal, StepSize )`



**Returns**     <listOf number> A list of *N* numbers starting with *StartVal* and changing with *StepSize*.  
**N**             <integer> The number of values to calculate for the list.  
**StartVal**     <number> The first value of the list.  
**StepSize**     <number> The quantity added to each successive element of the list.

**append( Seq1, ... )**

**Returns**     <listOf anything> A list derived from concatenating all arguments to the function.  
**Seq1, ...**     <listOf anything> A sequence of lists containing any datatype known to Alpha\_1.

**seq1( N )**

**Returns**     <listOf integer> A list of *N* positive integers starting at 1.  
**N**             <integer> The number of values to calculate for the list.

**seq0( N )**

**Returns**     <listOf integer> A list of *N* positive integers starting at 0.  
**N**             <integer> The number of values to calculate for the list.

For accessing the elements of a sequence, one can simply index them, as in

**Seq[2]**

where the first element has index 0. There are some other indexing functions which may sometimes be more natural.

**first( Seq )**

**Returns**     <anything> The first value in the list *Seq*.  
**Seq**           <listOf anything> A list whose members are any datatype known to Alpha\_1.

**second( Seq )**

**Returns**     <anything> The second value in the list *Seq*.  
**Seq**           <listOf anything> A list whose members are any datatype known to Alpha\_1.

**third( Seq )**

**Returns**     <anything> The third value in the list *Seq*.  
**Seq**           <listOf anything> A list whose members are any datatype known to Alpha\_1.

**fourth( Seq )**

**Returns**     <anything> The fourth value in the list *Seq*.  
**Seq**           <listOf anything> A list whose members are any datatype known to Alpha\_1.

**nth( Seq, N )**

**Returns**     <anything> The *N*th value in the list *Seq*, where the first value has index 0.

- Seq**      <listOf anything> A list whose members are any datatype known to Alpha\_1.
- N**        <integer> The index of the desired item. The first item has index 0.

## 7.2 Generic Operations

A few geometric operations are “universal” or “generic” in the sense that they can be applied to almost any object.

Any object with an implied orientation (lines, arcs, curves, surfaces, etc.) can have that orientation reversed by applying the **reverseObj** operator. This usually involves something like reversing the order that control points are listed, so it is a good idea to think about orientation early, especially if you tend to use constructions that depend on knowing the “first” or “last” point of an object.

**reverseObj( Obj )**

- Returns**    <anything> An object which is “flipped” with respect to *Obj*.
- Obj**        <anything> The object to be “flipped”. Note that **reverseObj** can be called with any datatype, but will only modify those with an implied orientation.

Any bounded geometric object (not lines or planes) may have a *bounding box* computed. The bounding box is the smallest rectangular box aligned with the coordinate axes which will contain the object. It is represented as two points. The minimum point has coordinates which are the minimum along each coordinate axis. The maximum point is computed similarly.

**bboxObj( Obj )**

- Returns**    <bBox> Calculate a bounding box for the indicated geometric object.
- Obj**        <object> This can be any object, however useful results will only be obtained from bounded geometric objects.

Another common geometric operation that can apply to a variety of objects is reflection through an axis.

**reflect( Obj, Axis )**

- Returns**    <object> Reflect the indicated object about *Axis*.
- Obj**        <object> Any geometric object can be reflected.
- Axis**       <point | line | plane> The object to reflect about. (The following ids may also be used: 'X', 'Y', 'Z', 'XY', 'XZ', 'YZ'.)

Finally, standard graphical transformations (scales, translates, and rotations) may be applied in a generic way. There are two ways to do this in **shape\_edit**: via the **instance** command, or via the **objTransform** command. Both of these are described fully in separate chapter (see chapter 9 [Transforming and Grouping Objects], page 143).

## 7.3 Points & Vectors

Alpha\_1 provides several types of points and vectors, for different geometric requirements. Some geometric operations are not meaningful on certain kinds of points or vectors. In cases where the operations are geometrically valid, the operations coerce the arguments to be compatible.

Points have position, but not length or direction. A point in space exists independently of any coordinate system which may be imposed, and thus is a coordinate-free object. However, we represent points using coordinates relative to a local or global coordinate system.

A Euclidean point in either 2D or 3D may be created, and is called an *e2Pt* or *e3Pt* respectively.

Vectors have length and direction, but not position. Like points, vectors exist independently of any coordinate system which may be imposed, and thus are also coordinate-free objects. Again, we represent vectors using coordinates relative to a local or global coordinate system.

A Cartesian vector in 2D or 3D is the most typical in modeling and is called an *r2Vec* or *r3Vec* respectively. A vector of dimension other than 2 or 3 is called an *rnVec*.

The simplest way to create points is to just specify the coordinates of the point.

There are constructor functions for each kind of point and vector discussed above. These constructors infer the dimension of the point from the number of arguments.

A Euclidean point in 2D or 3D (called an *e2Pt* or *e3Pt*) is created using *pt*.

*pt*( *X*, *Y* )

*Returns*     <*e2Pt*> Create a 2D Euclidean point with the given coordinates.

*X*, *Y*        <number> The X and Y coordinates of the point.

*pt*( *X*, *Y*, *Z* )

*Returns*     <*e3Pt*> Create a 3D Euclidean point with the given coordinates.

*X*, *Y*, *Z*     <number> The X, Y, and Z coordinates of the point.

Vectors of any dimension can be created by specifying the appropriate number of coordinates. The most common vectors, in 2D and 3D, are called *r2Vec* and *r3Vec*, while the general case of an n-dimensional vector is called an *rnVec*.

*vec*( *X*, *Y* )

*Returns*     <*r2Vec*> Create a 2D vector from the given values.

*X*, *Y*        <number> The X and Y components of the vector.

*vec*( *X*, *Y*, *Z* )

*Returns*     <*r3Vec*> Create a 3D vector from the given values.

*X*, *Y*, *Z*     <number> The X, Y, and Z components of the vector.

*vec*( *Num1*, ... )

*Returns*     <*anyVector*> Create an n-dimensional vector from n values.

*Num1*, ...    <number> Any number of values which are the components of the vector.  
The dimensionality of the vector is equal to the number of values given.

A set of operations is provided for examining the value of any coordinate in a constructed point or vector. This kind of operation is called an *extractor* function because it extracts a particular piece of information from a geometric entity. These operations check the type of the point or vector given as the argument and generate an error message if the requested coordinate does not exist for the given point or vector.

*ptX*( *Pt* )

*Returns*     <number> The value of the X component of the point.

*Pt*            <point> The point to examine.

**ptY( Pt )**

*Returns*     <number> The value of the Y component of the point.  
*Pt*            <point> The point to examine.

**ptZ( Pt )**

*Returns*     <number> The value of the Z component of the point.  
*Pt*            <e3Pt> The point to examine.

**vecX( Vec )**

*Returns*     <number> The value of the X component of the vector.  
*Vec*           <anyVector> The vector to examine.

**vecY( Vec )**

*Returns*     <number> The value of the Y component of the vector.  
*Vec*           <anyVector> The vector to examine. Note the vector must have a dimension of at least 2.

**vecZ( Vec )**

*Returns*     <number> The value of the Z component of the vector.  
*Vec*           <r3Vec | anyVector> The vector to examine. Note the vector must have a dimension of at least 3.

You may also simply index the coordinates of a point or vector, as in

`P1[0]`

or

`VecA[4]`

All of the extractors given above also work as *depositor* functions when used on the left hand side of an assignment operation. This means that they can be used to set the values of particular coordinates as well as to examine them. The following example illustrates this use:

`Pt1 := Pt( 1, 2, 0 );     % Construct an e3Pt.`

`ptZ( Pt1 ) := 25;        % Change the Z coordinate from 0 to 25.`

Coercion functions are used to add flexibility by converting whatever point or vector type is given to the desired type and dimensionality. Coercing any point or vector from 2D to 3D is done by adding a Z coordinate of 0. Coercing any point or vector from 3D to 2D is done by dropping the Z coordinate, regardless of its value. The effect is that 2D constructions are embedded within the XY-plane of the current 3D coordinate system. The coercion functions are named by the desired type of the result.

**e2( Pt )**

*Returns*     <e2Pt> Coerce the given point to a 2D Euclidean point.  
*Pt*           <point> A point of any kind.

**e3( Pt )**

*Returns*     <e3Pt> Coerce the given point to a 3D Euclidean point.  
*Pt*           <point> A point of any kind.

**r2( Vec )**

**Returns** <r2Vec> Coerce the given vector to a 2D vector.

**Vec** <anyVector> A vector of any kind.

**r3( Vec )**

**Returns** <r3Vec> Coerce the given vector to a 3D vector.

**Vec** <anyVector> A vector of any kind.

A set of *predicate* functions are provided for determining the types of points and vectors. A predicate function is one that asks a question which has a true or false answer. We can test whether any entity is a point using **pointP**. The operation returns a true value only if the entity is a point. More specific information about the type of a point can be obtained using **euclideanP** which returns a true value only if the argument is a euclidean point. (see section 7.8 [Circular Arcs], page 61 for an introduction to another kind of point, *projective* points.) A similar test for vectors, **vecP**, returns a true value only if the entity is a vector.

**pointP( Any )**

**Returns** <boolean> True if the argument is a point of any kind.

**Any** <anything> The data item to test.

**euclideanP( Any )**

**Returns** <boolean> True if the argument is a Euclidean point.

**Any** <anything> The data item to test.

**vecP( Any )**

**Returns** <boolean> True if the argument is a vector of any kind.

**Any** <anything> The data item to test.

We can also request information about the dimension of a point or vector. For euclidean points, the result of **ptDim** is the dimension (2 or 3) of the point, which is equal to the number of coordinates in the point. For projective points, the result is also the dimension (2 or 3), which is one less than the number of coordinates (because the W coordinate isn't an extra dimension). For vectors, **vecDim** returns the dimension of the vector. The **ptSize** and **vecSize** functions returns the actual number of coordinates in a point or vector. The **vecSize** function is the same as **vecDim**.

**ptDim( Pt )**

**Returns** <integer:2-3> Get the dimensionality of the point argument. Note for projective points the W coordinate is not part of the dimension.

**Pt** <point> The point to examine.

**vecDim( Vec )**

**Returns** <integer> Get the dimensionality of the vector argument.

**Vec** <anyVector> The vector to examine.

**ptSize( Pt )**

**Returns** <integer> The number of coordinates in the point.

**Pt** <point> The point to examine.

**vecSize( Vec )**

**Returns** <integer> Number of coordinates in the vector.

**Vec**            <vector> The vector to examine.

Several predefined points and vectors are often useful. The **Origin** is an **e2Pt** which most of the geometric construction operations will coerce to 3D as necessary. The vectors in the direction of the X axis and Y axis, **XDir** and **YDir**, are of type **r2Vec** (but also coerced by the construction operations where necessary), while the vector in the direction of the Z axis, **ZDir**, is an **r3Vec**.

#### **Origin**

**<e2Pt>**        The value of the origin. Note that although this is an **e2Pt** most functions will coerce it when necessary.

#### **XDir**

**<r2Vec>**        A vector pointing in the direction of the positive X axis.

#### **YDir**

**<r2Vec>**        A vector pointing in the direction of the positive Y axis.

#### **ZDir**

**<r3Vec>**        A vector pointing in the direction of the positive Z axis.

Although it is not a predefined vector, a vector of all zeros is often useful, and can be created with **vecOrigin**.

**vecOrigin( Dim )**

**Returns**        <anyVector> A vector of zeros.

**Dim**            <integer> The dimensionality of the desired zero vector.

## 7.4 Operations on Points & Vectors

A number of operations can be performed on vector objects, for example adding two of them. Euclidean points may not be added, but they may be subtracted (yielding a vector). Some operations are defined which involve euclidean points and vectors, for example adding a vector to a euclidean point (yielding a euclidean point).

For the remainder of this section, "point" will refer to a euclidean point, either 2D or 3D. The operations will coerce 2D euclidean points to 3D when necessary.

The angle of a vector, is computed in degrees counter-clockwise from 0 to the right. We can construct a vector at a given angle (**vecAtAngle**), find out the angle or length of a vector (**vecAngle**, **vecLength**), or normalize it by dividing each coordinate by the length of the vector (**unitVec**).

**vecAtAngle( Angle )**

**Returns**        <2rVec> Construct a vector at the given angle from positive X.

**Angle**            <number> The angle of the vector, positive angles rotate counter-clockwise from the right.

**vecAngle( Vec )**

**Returns**        <number> Return the angle of the given vector in degrees from the X axis, positive values indicate counter-clockwise rotation.

**Vec**            <anyVector> The vector to test. Note the vector must have a dimension of at least 2.

**vecLength( Vec )**

**Returns**     <number> Calculate the length of the given vector.  
**Vec**           <anyVector> The vector to examine.

**unitVec( Vec )**

**Returns**     <anyVector> Normalize the given vector.  
**Vec**           <anyVector> The vector to normalize.

The distance between two points is given by **distPtPt**.

**distPtPt( Pt1, Pt2 )**

**Returns**     <number> Calculate the distance between two points.  
**Pt1, Pt2**    <point> Two points to check. The distance is always a positive value.

Two vectors may be added or subtracted with **vecPlus** or **vecMinus**, producing the vector sum or difference. If the vectors are of different dimensions, the lower dimension one is coerced to the dimension of the other by adding coordinates with value 0 as needed. A vector may be used to construct a point offset from a given point with **ptOffset**. The vector is added to the given point, producing the offset point. The dimension of the point or the vector is coerced to 3D if either is already 3D. A variant operation, **ptScaledOffset**, allows the vector to be scaled by a given factor before adding it to the point.

**vecPlus( V1, V2 )**

**Returns**     <anyVector> Perform a component-wise sum of the two vectors. The result will have the same dimension as the larger of the two arguments.  
**V1, V2**       <anyVector> The two vectors to sum. They need not be the same dimension.

**vecMinus( V1, V2 )**

**Returns**     <anyVector> Perform a component-wise difference of the two vectors. The result will have the same dimension as the larger of the two arguments.  
**V1, V2**       <anyVector> The two vectors to subtract, *Vec1* minus *Vec2*.

**ptOffset( Pt, Vec )**

**Returns**     <euclidPoint> Create a point offset from *Pt* by the vector *Vec*. If the dimensions of the point and vector are different, the larger dimension is used.  
**Pt**            <euclidPoint> The reference point to offset from.  
**Vec**           <geomVector> The vector to offset by.

**ptScaledOffset( Pt, Vec, Factor )**

**Returns**     <euclidPoint> Create a point offset from a reference by *Vec\*Factor*.  
**Pt**            <euclidPoint> The reference point to offset from.  
**Vec**           <geomVector> The direction to offset in.  
**Factor**        <number> The scale factor to use in offsetting.

Although points may not be added because it is geometrically meaningless, they may be subtracted to yield a vector which describes the direction from one to the other using one of three equivalent functions. **VecOffset** and **vecFrom2Pts** are simply alternate names for the same operation. The direction of the resulting vector is from the *BasePt* to the *EndPt*. It is sometimes more convenient

to think of the operation as subtraction of two points, in which case **ptMinus** again does the same computation, but the order of the arguments is reversed, yielding a vector in the direction from *Pt2* to *Pt1*.

**vecOffset**( *BasePt*, *EndPt* ) or

**vecFrom2Pts**( *BasePt*, *EndPt* )

**Returns** <geomVector> Calculate the component-wise difference of two points.  
*BasePt*, *EndPt*

<euclidPoint> The two endpoints of the vector. These may be of different dimensions and the larger dimension is used.

**ptMinus**( *EndPt*, *BasePt* )

**Returns** <geomVector> Calculate the component-wise difference of two points.  
*EndPt*, *BasePt*

<euclidPoint> The two endpoints of the vector. Note that the order of these arguments is reversed from that of **vecOffset**.

Scalar multiplication is allowed for vectors. The **vecScale** operation results in a vector scaled by the specified amount. It is geometrically meaningless to scale points, so no operation is provided for this. (To scale objects containing points see chapter 9 [Transforming and Grouping Objects], page 143.)

**vecScale**( *Vec*, *Factor* )

**Returns** <anyVector> Perform a component-wise scaling of a vector.

*Vec* <anyVector> The vector to scale.

*Factor* <number> The scale factor to use.

Several kinds of blending of points and vectors are allowed. The simplest and most often used blending is just a linear interpolation between two points. Any point on the line connecting the two points can be produced by choosing an appropriate parameter value for **ptInterp**. If *Param* is 0, the result is just *Pt1*. If *Param* is 0.5, the result is the midpoint of the line segment between *Pt1* and *Pt2*. If *Param* is 1, the result is *Pt2*. A similar interpolation operation, **vecInterp** is provided for vectors.

**ptInterp**( *Pt1*, *Pt2*, *Param* )

**Returns** <euclidPoint> Interpolate a new point from two reference points.

*Pt1*, *Pt2* <euclidPoint> The two points to use, *Pt1* is the "zero point", *Pt2* is the "one point".

*Param* <number> The blending parameter. A value of 0.0 results in *Pt1*, and value of 1.0 results in *Pt2*.

**vecInterp**( *V1*, *V2*, *Param* )

**Returns** <anyVector> Linearly interpolate between two vectors.

*V1*, *V2* <anyVector> The two vectors to interpolate between.

*Param* <number> The blending factor. If the value is 0.0 then *V1* results, if the value is 1.0 *V2* results.

A more general blending is also supported for points or vectors. Currently only convex blends are allowed. The coefficient associated with each point must be in the range [0,1] and the sum of all the coefficients must be equal to 1. The number of coefficients must match the number of points



or vectors being blended.

**ptBlend( PtList, CoeffList )**

**Returns** <euclidPoint> Perform a convex blend of a set of points and return a new point.

**PtList** <listOf euclidPoint> This is the list of points to blend. The dimension of the resulting point is the same as the first point of this list.

**CoeffList** <listOf number> This is the list of weight factors each point is to receive.

**vecBlend( VecList, CoeffList )**

**Returns** <anyVector> Perform a convex blend of a set of vectors and return a new vector.

**VecList** <listOf anyVector> A list of vectors to blend. The resulting vector will have a dimension the same as the first vector in the list.

**CoeffList** <listOf number> A list of weight factors each vector is to receive.

Vector operations of dot product and cross product are also provided. Dot product, **dotProd**, may be applied to vectors of any dimension. Cross product is geometrically meaningful only for R3 vectors, or R2 vectors which are coerced to 3D for the computation. The result of **crossProd** is always an r3Vec.

**dotProd( V1, V2 )**

**Returns** <anyVector> Calculate the dot product of two vectors. The result will have the dimension of the largest argument.

**V1, V2** <anyVector> The two vectors to product.

**crossProd( V1, V2 )**

**Returns** <geomVector> Calculate the cross product of two vectors. The result will have the dimension of the largest argument.

**V1, V2** <geomVector> The two vectors to cross product.

As an example of the point and vector operations introduced so far, consider constructing the points which form a square of size **SquareSize** in 2D, with lower left corner at the origin.

```
% Lower left corner will be at the origin.
```

```
Pt1 := Origin;
```

```
% Lower right corner. Add vector in direction of X axis, and
% scaled to desired length to the first point.
```

```
Pt2 := ptScaledOffset( Pt1, Xdir, SquareSize );
```

```
% Similar construction for top right and left corners.
```

```
Pt3 := ptScaledOffset( Pt2, Ydir, SquareSize );
```

```
Pt4 := ptScaledOffset( Pt3, Xdir, -SquareSize );
```

```
% The center of gravity of the square is the average of the four
% points.
```

```
CenterOfGravity := ptBlend( list( Pt1, Pt2, Pt3, Pt4 ),
                             nTimes( 4, 0.25 ) );
```

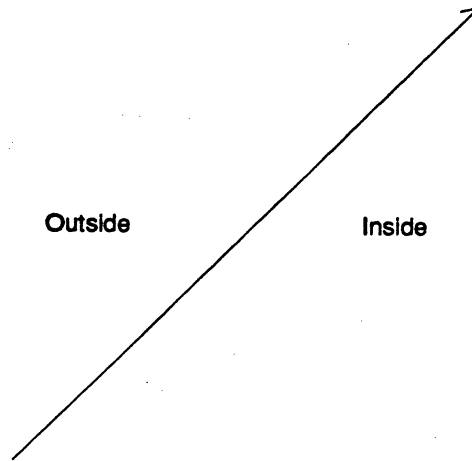


Figure 7-1: Orientation of Constructed Lines

## 7.5 Lines

Lines are provided in the basic plane geometry sense, and are often a nice way to express dimensions for use in constructing points and arcs. An Alpha\_1 line has infinite extent in both directions. Two types of lines are provided depending on dimension. A 2D line is called an *e2Line*, and a 3D line is called an *e3Line*.

E2Lines are represented by a line equation in the plane. The representation for e3Lines is more complicated and will not be discussed here. Lines have a direction, which can be extracted by the function `dirOfLine` described below. For e2Lines there is also a concept of “inside”, related to the line’s direction. By convention, the inside is to the right in the XY-plane if you look in the direction of the line (Figure 7-1).

Points used in the construction of lines or extracted from lines are coerced to be euclidean. (Again, “point” below will mean a euclidean point unless otherwise specified.)

There are several ways to construct lines. Vertical and horizontal lines, parallel to the X and Y axes can be generated easily. For vertical lines constructed with `lineVertical`, the “inside” is to the right of the line. For horizontal lines constructed with `lineHorizontal`, the “inside” is below the line. In other words, the direction of a `lineVertical` is up and the direction of a `lineHorizontal` is to the right. The direction of a line can be reversed using `reverseObj`.

`lineVertical( Xoffset )`

**Returns**     `<e2Line>` Create a 2D line with the indicated X coordinate parallel to the Y axis.

**Xoffset**     `<number>` The desired X coordinate of the vertical line.

**lineHorizontal( Yoffset )**

**Returns** <e2Line> Create a 2D line with the indicated Y coordinate parallel to the X axis.

**Yoffset** <number> The desired Y coordinate of the horizontal line.

A line can be specified by a point and an angle with **linePtAngle**. The line goes through the given point, and has direction specified by the angle. The angle is in degrees, measured counter-clockwise from the X axis. If the point argument is an e2 point, the line will be an e2Line. If it is an e3 point, the resulting line will be an e3Line parallel to the XY-plane. We can also construct a line through a point, in a direction given by a vector (**linePtVec**), or a line passing through two points, with direction from the first point to the second (**lineThru2Pts**).

**linePtAngle( Pt, Angle )**

**Returns** <line> Create a line through the given point at the indicated angle. If an e3Pt is given the line will be a 3D line, otherwise it will be 2D.

**Pt** <point> The point which the line should pass through.

**Angle** <number> The angle of the line in degrees counter-clockwise from the positive X axis.

**linePtVec( Pt, Vec )**

**Returns** <line> Create a line from the given point and vector. If either the point or vector are 3D, then the line will be 3D.

**Pt** <point> The point the line should pass through.

**Vec** <geomVector> A vector in the direction of the line's positive end.

**lineThru2Pts( Pt1, Pt2 )**

**Returns** <line> Create a line through the two points. If either point is 3D, then the line will be 3D.

**Pt1, Pt2** <point> The points defining the line. The positive direction of the line is from Pt1 to Pt2.

Lines can also be specified relative to other lines. The operation **linePtParallel** produces a line through the given point, parallel to the given line, while **lineOffsetFromLine** generates a line offset from the given line. The sign of the offset is significant, with positive values producing lines to the "inside" or right of the original line. The **LineOffsetFromLine** operation is only defined for e2Lines, since the notion of "inside" does not exist for e3Lines.

**linePtParallel( Pt, Line )**

**Returns** <line> Create a line through a point parallel to another line.

**Pt** <point> The point the line should pass through.

**Line** <line> The line it should be parallel to.

**lineOffsetFromLine( Offset, Line )**

**Returns** <e2Line> Create a line offset from another line. The reference line must lie in the XY-plane.

**Offset** <number> The offset distance. A positive number offsets to the right and perpendicular to the reference line.

**Line** <e2Line> The reference line (which must lie in the XY-plane).

The line of intersection of two planes is computed by **lineIntersect2Planes**. The resulting line

satisfies the sign convention that its direction is the same as the cross product of the two plane normals.

**lineIntersect2Planes( Pl1, Pl2 )**

**Returns** <e3Line> Create a line by intersecting two planes.  
**Pl1, Pl2** <plane> The two planes to intersect.

The line from a point tangent to a circle is constructed with **linePtCircle**. The point must be outside the circle for this to be valid. A flag argument determines which of the two lines will be returned: T for the one which comes tangent to the counter-clockwise side of the circle (so the circle is on the left of the line), and Nil for the other.

**linePtCircle( Pt, Cir, CcwFlag )**

**Returns** <line> Create a line through a point and tangent to a circle. If either the point or circle are 3D, then the line will be 3D.  
**Pt** <point> The point the line should pass through.  
**Cir** <circle> The circle the line should be tangent to.  
**CcwFlag** <boolean> If true, the line will be tangent to the counter-clockwise side of the circle.

A line tangent to two circles is constructed using **lineTan2Circles**. The line goes in the direction from the first circle to the second. Since there are 4 lines tangent to a given pair of circles (8, counting the orientation of the line), a flag is given with each circle to indicate whether the line is tangent to the circle going clockwise or counter-clockwise (T indicates ccw.) If the flags are different (one T and the other Nil) the line will cross between the two circles. The circles must not overlap in this case. If the flags are both T the circles will be to the left of the line, and if both Nil they will be on the right. One circle must not be completely inside the other in this case.

**lineTan2Circles( Cir1, Ccw1, Cir2, Ccw2 )**

**Returns** <line> Construct a line tangent to two circles.  
**Cir1, Cir2** <circle> The two circles which the line must be tangent to. These circles must be coplanar.  
**Ccw1, Ccw2** <boolean> These flags indicate where the tangent should touch each circle. If true the line should touch the circle going counter-clockwise.

If a line needs to be normalized (if one is constructed by some other mechanism than these operations, for example), use **normalizeLine**. It returns a normalized representation of the line.

**normalizeLine( Line )**

**Returns** <line> Normalize a line.  
**Line** <line> The line to normalize.

Coercion functions for lines are available for converting whatever line type is given to the desired type. The coercion functions are named by the desired type of the result. A line parallel to the Z axis does not project to a line in the XY-plane. It is, therefore, an error to call **e3** on such a line.

**e2( Line )**

**Returns** <e2Line> Convert the given line to a 2D line.  
**Line** <line> Any line (except one parallel to the Z axis).

**e3( Line )**

**Returns** <e3Line> Convert the given line to a 3D line.  
**Line** <line> Any line.

We can test whether any entity is a line using the **lineP** predicate. The operation returns a true value only if the entity is a line. Two predicates, **linesParallelP** and **linesSkewP** are available for determining the relationship between two lines.

**lineP**( Any )

**Returns** <boolean> Determine if a given item is a line.  
**Any** <anything> Any datatype known to Alpha\_1.

**linesParallelP**( Ln1, Ln2 )

**Returns** <boolean> Determine if the given two lines are parallel.  
**Ln1, Ln2** <line> The two lines to test.

**linesSkewP**( Ln1, Ln2 )

**Returns** <boolean> Determine if two lines are skew.  
**Ln1, Ln2** <line> The two lines to test.

A number of extraction operations for lines are provided. The direction of a line (an r2Vec or r3Vec depending on line type) is obtained with **dirOfLine**. The vector result of **dirPerpLine** points to the "inside", or right of the line, but is only defined for e2Lines. A direction lying in a plane and perpendicular to any line is computed by **dirPerpLineInPlane**. If it is called on an e2Line with the positively oriented XY-plane (that is, with normal vector **ZDir** ), it will return the same result as **DirPerpLine**. The angle of a line (angle of its direction vector) is computed in degrees counter-clockwise from 0 to the right. This operation is only defined for e2Lines.

**dirOfLine**( Line )

**Returns** <geomVector> Determine the direction of the given line.  
**Line** <line> Any line.

**dirPerpLine**( Line )

**Returns** <geomVector> Calculate the vector perpendicular to a line. The vector will point to the "inside" of the line.  
**Line** <e2Line> The reference line (must be in the XY-plane).

**dirPerpLineInPlane**( Line, Plane )

**Returns** <geomVector> Calculate a direction vector perpendicular to a given line and lying in the indicated plane.  
**Line** <line> The reference line.  
**Plane** <plane> The plane in which the line should lie.

**angleOfLine**( Line )

**Returns** <number> Determine the angle of a line in degrees from the positive X axis.  
**Line** <e2Line> Any line lying in the XY-plane.

Sometimes it is useful to produce an arbitrary point on a line in a way that will always avoid division by zero. The function **ptOnLine** provides that capability. A point on a line with a specified coordinate value is given by **ptOnLineWithX**, **ptOnLineWithY**, or **ptOnLineWithZ**. It is an error if all (or none) of the points on the line have the specified coordinate value.

**ptOnLine( Line )**

**Returns** <euclidPoint> Generate an arbitrary point on a given line.

**Line** <line> Any 2D or 3D line.

**ptOnLineWithX( Line, XCoord )**

**Returns** <euclidPoint> Calculate a point on a given line with a particular X coordinate.

**Line** <line> The reference line.

**XCoord** <number> The desired X coordinate of the point.

**ptOnLineWithY( Line, YCoord )**

**Returns** <euclidPoint> Calculate a point on a given line with a particular Y coordinate.

**Line** <line> The reference line.

**YCoord** <number> The desired Y coordinate of the point.

**ptOnLineWithZ( Line, ZCoord )**

**Returns** <e3Pt> Calculate a point on a given line with a particular Z coordinate.

**Line** <line> The reference line.

**ZCoord** <number> The desired Z coordinate of the point.

The point at the intersection of two lines is given by **ptIntersect2Lines**. The **ptOnLineNearestLine** function computes the point on one line nearest another line, and **ptNearest2Lines** calculates the point nearest two lines. For either of these two functions, if the lines intersect the intersection point will be returned. In either function it is an error for the lines to be parallel.

**ptIntersect2Lines( Ln1, Ln2 )**

**Returns** <euclidPoint> Calculate the intersection point of two lines. The lines need not be coplanar.

**Ln1, Ln2** <line> The lines to intersect.

**ptOnLineNearestLine( Ln1, Ln2 )**

**Returns** <euclidPoint> Calculate the point on one line nearest another line.

**Ln1, Ln2** <line> The two lines to use.

**ptNearest2Lines( Ln1, Ln2 )**

**Returns** <euclidPoint> Calculate a point which is the midpoint of the shortest line segment between two lines. This point is as near as possible to both lines.

**Ln1, Ln2** <line> The two lines to use.

The shortest line segment joining two lines is given by **segBetween2Lines**. This segment is represented as a polyline with two points. For skew lines this shortest segment is well defined. For intersecting lines it is a degenerate line segment. And for parallel lines there is no unique shortest segment. In this case it returns one of the infinitely many possible solutions.

**segBetween2Lines( Ln1, Ln2 )**

**Returns** <polyline> Calculate the shortest line segment connecting two lines.

**Ln1, Ln2** <line> The two lines to use.

The **projectPtOntoLine** operation performs perpendicular projection of a point onto a line.

**projectPtOntoLine( Pt, Line )**

**Returns** <euclidPoint> Calculate a point on a given line which is the perpendicular projection of a given point.

**Pt** <point> The point to project.

**Line** <line> The reference line to project onto.

The (unsigned) distance from any point to a line is given by **distPtLine**. For "signed" distance, use **signedDistPtLine**. The third argument is optional for **e2Lines**, but required for **e3Lines**. If omitted, the function returns a positive number if the point is to the right of the line ("inside"), and negative if it is to the left ("outside"). With the third argument, which works for all lines, the signed distance is positive if the reference vector points from the line toward the point. The minimum distance between two lines is computed with **distLineLine**.

**distPtLine( Pt, Line )**

**Returns** <number> Calculate the unsigned distance from a point to a line.

**Pt** <euclidPoint> The reference point.

**Line** <line> The target line.

**signedDistPtLine( Pt, Line, Vec )**

**Returns** <number> Calculate the signed distance from the point to the line.

**Pt** <euclidPoint> The reference point.

**Line** <line> The target line.

**Vec** <opt geomVector> The vector defining positive distances. If the vector points toward the point the distance will be positive. This argument is optional for **e2Lines** and required for **e3Lines**.

**distLineLine( Ln1, Ln2 )**

**Returns** <number> Compute the minimum distance between to (possibly parallel) lines.

**Ln1, Ln2** <line> The two lines to use.

Three predefined lines are available for use if desired. They are the lines which go in the direction of the X, Y, and Z axes and which intersect at the origin:

**XAxis**

<e2Line> A line defining the X axis.

**YAxis**

<e2Line> A line defining the Y axis.

**ZAxis**

<e3Line> A line defining the Z axis.

As an example of using the line constructors, consider defining the four lines which frame a diamond shape centered on the origin (Figure 7-2).

The inside of each line should be towards the diamond.

```
% Base point is lowest point.
```

```
BasePt := pt( 0, -0.8 );
```

```
% Lower left line.
```

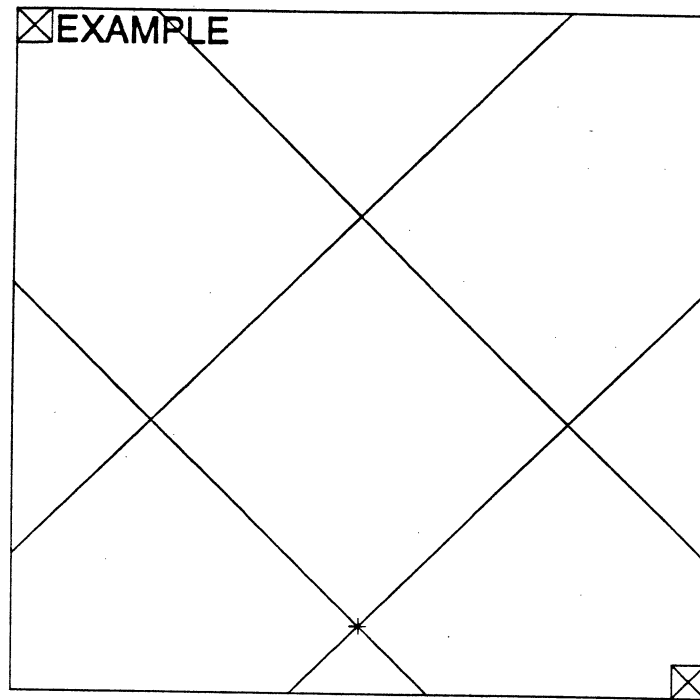


Figure 7-2: Example of Line Construction

```

LLLine := linePtAngle( BasePt, 135 );
% Lower right line.
LRLine := reverseObj linePtVec( BasePt, dirPerpLine LLLine );

% The upper lines are just offsets of the lower ones, with the
% directions reversed.
URLine := reverseObj linePtParallel( pt( 0, 0.8 ), LLLine );
ULLine := reverseObj linePtParallel( pt( 0, 0.8 ), LRline );

```

## 7.6 Polylines & Polygons

Polylines and polygons are made up of sequences of points. The points in a polyline, also called vertices, are connected by line segments. If the first point is repeated at the end of the polyline, the polyline is closed. A polygon is similar, except that polygons are always implicitly closed without having to repeat the first vertex.

Polylines mostly exist for communicating with line-drawing display devices, so the operations provided are minimal. A polyline is constructed by specifying a list of points for the **polyline** function. Polygons are constructed the same way, using the **polygon** function. Polylines and polygons have orientation (for more information see chapter 5 [Orientation Conventions], page 29). The orientation can be reversed using **reverseObj**. Finally, you can convert a polygon into a polyline with **polylineFromPolygon**.

```
polyline( PtList )
```



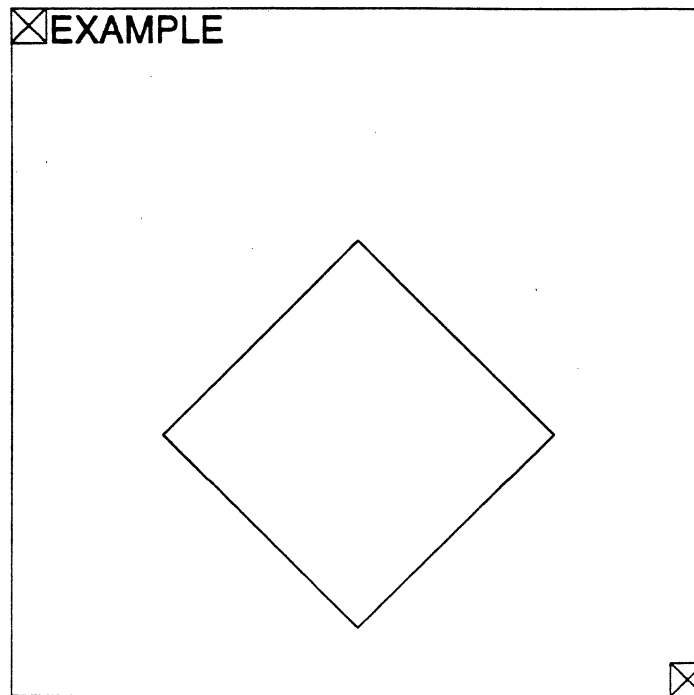


Figure 7-3: Example of Polyline Construction

*Returns* <polyline> Generate a polyline from a list of points.  
*PtList* <listOf point> A list of points to use in constructing the polyline.  
**polyline**( *PtList* )

*Returns* <polygon> Generate a polygon from a list of points.  
*PtList* <listOf point> A list of points to use in constructing the polygon.  
**polylineFromPolygon**( *Poly* )

*Returns* <polyline> Coerce a polygon to a polyline.  
*Poly* <polygon> The polygon to coerce.

A polyline (Figure 7-3) which describes the outline of the diamond constructed in the last section could be defined as:

```
DiamondLine := polyline(
    list( ptIntersect2Lines( LRLine, URLine ),
          ptIntersect2Lines( LLLine, LRLine ),
          ptIntersect2Lines( ULLine, LLLine ),
          ptIntersect2Lines( URLine, ULLine ),
          ptIntersect2Lines( LRLine, URLine ) ));
```

## 7.7 Planes

A plane is represented in **shape\_edit** by its normal and distance from the origin. More precisely,

the plane equation  $Ax + By + Cz + D$  equals zero if the point  $(x, y, z)$  lies in the plane. The (unit) normal is defined by the  $A, B, C$  components and  $D$  is the negative of the offset of the plane from the origin. All the plane construction operations construct these normalized plane equations. The operations which reference planes assume that they are normalized. The positive side or "inside" of a plane is the side the normal points toward.

The basic plane construction, **planeThruPtWithNormal** uses a given point in the plane and a normal vector to the plane to construct the plane. Planes can also be constructed as offsets from a given plane with **planeOffsetByDelta**. Positive offsets going in the direction of the normal to the given plane. The **planeThruPtAndLine** operation produces a plane containing a point and a line. The resulting plane satisfies the sign convention that its normal is in the direction of the cross product of the vector from the point to its projection on the line with the direction of the line. A plane may be constructed to contain three points with **planeThru3Pts**. The resulting plane satisfies the following sign convention: If the fingers of the right hand follow the points in order, the plane normal will be in the direction of the thumb.

**planeThruPtWithNormal**( *Point*, *Normal* )

*Returns*     <plane> Construct a plane given a point and a normal.  
*Point*       <point> A reference point the plane should pass through.  
*Normal*      <geomVector> The normal to the plane.

**planeOffsetByDelta**( *Plane*, *Delta* )

*Returns*     <plane> Construct a plane offset from another plane.  
*Plane*       <plane> The reference plane to offset from.  
*Delta*       <number> A distance to offset, positive values offset in the direction of the plane normal.

**planeThruPtAndLine**( *Point*, *Line* )

*Returns*     <plane> Construct a plane through a point and a line.  
*Point*       <point> A point not on *Line*.  
*Line*       <line> A reference line.

**planeThru3Pts**( *P1*, *P2*, *P3* )

*Returns*     <plane> Construct a plane through three points. The plane normal follows the right-hand rule.

*P1, P2, P3*     <point> Three non-collinear points.

A plane may be constructed to contain two intersecting or parallel lines using **planeThru2Lines**. The normal to the resulting plane will be the cross product of the two line direction vectors (if the lines are not parallel). To construct a plane through one line parallel to a second line, use **planeThruLineParallelToLine**. If the lines intersect this has the same result as **planeThru2Lines**. For parallel or skew lines the resulting plane's normal will point toward the line which the plane does not contain.

**planeThru2Lines**( *Ln1*, *Ln2* )

*Returns*     <plane> Construct a plane through two parallel or intersecting lines.  
*Ln1, Ln2*    <line> The two lines to use.

**planeThruLineParallelToLine**( *Containing*, *ParallelTo* )

**Returns**    <plane> Construct a plane containing one line and parallel to another line.  
**Containing, ParallelTo**  
               <line> Two lines to use.

If a plane needs to be normalized (e.g., if one is constructed by some other mechanism than these operations), use **normalizePlane**. The direction of the plane normal (and hence the notion of which is the positive side of the plane) is reversed using the **reverseObj** operator.

**normalizePlane( Plane )**

**Returns**    <plane> Normalize a plane.  
**Plane**       <plane> The plane to normalize.

Some extraction operations for accessing information about planes are available. We can determine the normal of a plane using **planeNormal**, and the signed distance from a point to a plane using **signedDistPtPlane**. The **angleFrom2Planes** operation computes the angle between two planes. The point at the intersection of three planes is given by **ptFrom3Planes**. Finally, the point of intersection of a line and plane is given by **ptIntersectLineAndPlane**.

**planeNormal( Plane )**

**Returns**    <r3Vec> The normal to the given plane.  
**Plane**       <plane> The plane to examine.

**signedDistPtPlane( Point, Plane )**

**Returns**    <number> Calculate the signed distance from the point to the plane. A positive distance means the plane normal points toward the point.  
**Point**       <euclidPoint> The reference point.  
**Plane**       <plane> The target plane.

**angleFrom2Planes( Pl1, Pl2 )**

**Returns**    <number> Calculate the angle between two planes.  
**Pl1, Pl2**    <plane> The two planes to examine.

**ptFrom3Planes( Plane1, Plane2, Plane3 )**

**Returns**    <e3Pt> Calculate the intersection point of three planes.  
**Plane1, Plane2, Plane3**  
               <plane> The planes to intersect.

**ptIntersectLineAndPlane( Line, Plane )**

**Returns**    <e3Pt> Calculate the intersection point of a line and a plane.  
**Line**        <line> The reference line.  
**Plane**       <plane> The plane to intersect.

It is often useful to project various other geometric entities onto planes. A point may be projected onto a plane in any given direction with **ptProjPtDirPlane**. A vector is projected onto a plane in the normal direction using **vecProjVecOntoPlane**. Although curves have not yet been discussed, they also can be projected onto planes using **crvFromCrvProjOntoPlane**. If the projector is a vector, the projection is parallel to that vector direction. If the projector is a point, each defining point of the curve is point projected onto the plane along the line between it and the projector.

**ptProjPtDirPlane( Point, Dir, Plane )**

**Returns**    <euclidPoint> Project a point onto a plane along a given vector.

**Point**      <euclidPoint> The point to project.  
**Dir**        <geomVector> The vector along which to project it.  
**Plane**      <plane> The target plane on which the resulting point will lie.  
**vecProjVecOntoPlane( Vec, Plane )**  
**Returns**    <geomVector> Project a vector onto a plane in the normal direction.  
**Vec**        <geomVector> The vector to project.  
**Plane**      <plane> The target plane to project onto.  
**crvFromCrvProjOntoPlane( Curve, Projector, Plane )**  
**Returns**    <curve> Project a curve onto a plane. The projection can be either along a vector or through a point.  
**Curve**      <curve> The curve to project.  
**Projector**   <geomVector | euclidPoint> The reference vector to project along or point to project through.  
**Plane**      <plane> The target plane to project against.

## 7.8 Circular Arcs

Circular arcs are represented by the rational quadratic B-spline control polygon of the arc. The defining points of an arc (and of a rational B-spline) are projective points, rather than the euclidean points. For more information see section 7.3 [Points & Vectors], page 43 and see section 7.4 [Operations on Points & Vectors], page 47.

### Projective Points

A projective point may be either 2D or 3D, and is called a *p2Pt* or *p3Pt*, respectively. Projective points are usually only meaningful when embedded in arcs, spline curves, or spline surfaces. A *p2Pt* has X, Y, and W (or homogeneous) coordinates, while a *p3Pt* has X, Y, Z, and W coordinates. The coordinate extractors (**ptX**, **ptY**, **ptZ**) which operate on euclidean points also operate on projective points. In addition, **ptW** will extract the homogeneous (W) coordinate.

**ptW( Pt )**

**Returns**    <number> Extract the W component of the projective point.  
**Pt**        <projPoint> The projective point to examine.

If a projective point is to be coerced to a euclidean point (using the *e3* coercion operation, for example), the projection is done by dividing each coordinate by the W coordinate and then dropping the W coordinate. A euclidean point is coerced to a projective point by adding a W coordinate of 1. (Projecting this resulting point back to a euclidean point yields the original point.) The effect is that euclidean spaces are embedded in their projective counterparts at *W*=1. The coercion operations which produce projective points are *p2* and *p3*. It is possible to create projective points by specifying the coordinates using **projPt**. The projective analogue to the **euclideanP** operation, **projectiveP**, returns a true value only if the argument is a projective point.

**p2( Pt )**

**Returns**    <p2Pt> Coerce a point to be a 2D projective point.  
**Pt**        <point> The point to coerce.

**p3( Pt )**

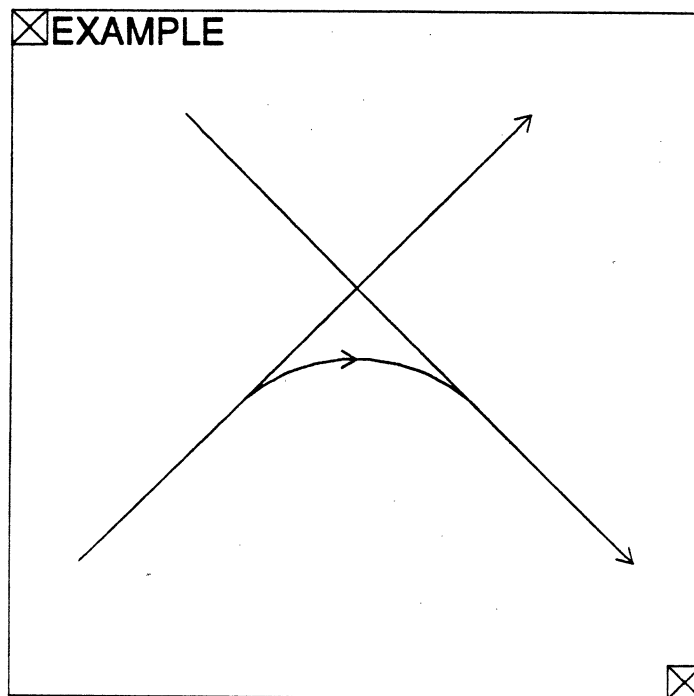


Figure 7-4: Orientation of Lines and Arc in Arc Construction

*Returns*     <p3Pt> Coerce a point to be a 3D projective point.

*Pt*            <point> The point to coerce.

**projPt( X, Y, W )**

*Returns*     <p2Pt> Create a 2D projective point.

*X, Y, W*      <number> The components of the point.

**projPt( X, Y, Z, W )**

*Returns*     <p3Pt> Create a 3D projective point.

*X, Y, Z, W*   <number> The components of the point.

**projectiveP( Any )**

*Returns*     <boolean> Determine if an item is a projective point.

*Any*          <anything> An datatype known to Alpha\_1.

### Arc Constructors

All of the operations that construct arcs (except **arcTan3Lines** and **otherArc**) produce valid arcs only for angles less than 180 degrees. This convention makes it clear which of several possible arcs will be constructed in many of the construction operations described below.

Arcs have an implied direction from the first endpoint to the second. In the case of constructing an arc from two lines, the first endpoint will lie on the first line and the arc will proceed from there in the same direction as the line. The arc will come in tangent to the second line, in the direction of the second line, at its second endpoint (Figure 7-4).

It may be necessary to reverse the direction of some constructed lines to achieve the desired arc. The direction of the arc itself will become relevant when chaining series of arcs together to form curves. As for lines and planes, the direction of an arc can be reversed with the **reverseObj** operation.

We can construct an arc from a radius and two tangent lines with **arcRadTan2Lines**, or from three lines with **arcTan3Lines**. Unlike most of the other arc constructions, **arcTan3Lines** generates arcs that subtend any angle less than 360 degrees.

**arcRadTan2Lines**( *Radius, From, To* )

*Returns*     <arc> Create an arc with given radius tangent to two lines.

*Radius*     <number> The radius of the resulting arc.

*From, To*    <line> The start of the arc will be tangent to *From*, the end will be tangent to *To*.

**arcTan3Lines**( *Start, Middle, End* )

*Returns*     <arc> Create an arc tangent to three lines.

*Start, Middle, End*

             <line> The lines the arc will be tangent to.

Given three points on the arc, we can construct the arc with the given endpoints which goes through the specified middle point with **arcThru3Pts**, or we can use **arcEndCenterEnd** given two endpoints and the center of the circle.

**arcThru3Pts**( *EndPt1, MiddlePt, EndPt2* )

*Returns*     <arc> Create an arc which begins and ends at two given points and passes through a third.

*EndPt1, MiddlePt, EndPt2*

             <euclidPoint> The three points defining the arc.

**arcEndCenterEnd**( *Start, CenterPt, End* )

*Returns*     <arc> Construct an arc with the given end points and the indicated center.

*Start, End*

             <euclidPoint> The end points of the arc.

*CenterPt*    <euclidPoint> The center of the arc.

To construct an arc starting at a point and tangent to two specified lines use **arcEndTan2Lines**. The point must lie on the first line, and the two lines must intersect. The *Obtuse* flag governs whether to construct the arc subtending an acute or obtuse angle: if Nil, the arc will be acute; if T it will be obtuse. If the two lines are perpendicular, neither arc will be obtuse nor acute. In this case, if *Obtuse* is Nil, the arc's direction where it meets *To* will be the same as that of *To*. If T, the directions will be opposing.

**arcEndTan2Lines**( *EndPt, From, To, Obtuse* )

*Returns*     <arc> Construct an arc with the given end point and tangent to two lines.

*EndPt*       <euclidPoint> The starting point of the arc.

*From, To*    <line> The two lines the arc should be tangent to. The point must lie on the first line.

*Obtuse*       <boolean> If true, construct an arc subtending the obtuse angle between the two lines. If the lines are perpendicular the arc will travel in the opposite direction from *To*.

The **arcRadTanToCircleAndLine** operation constructs an arc of given radius which is tangent to

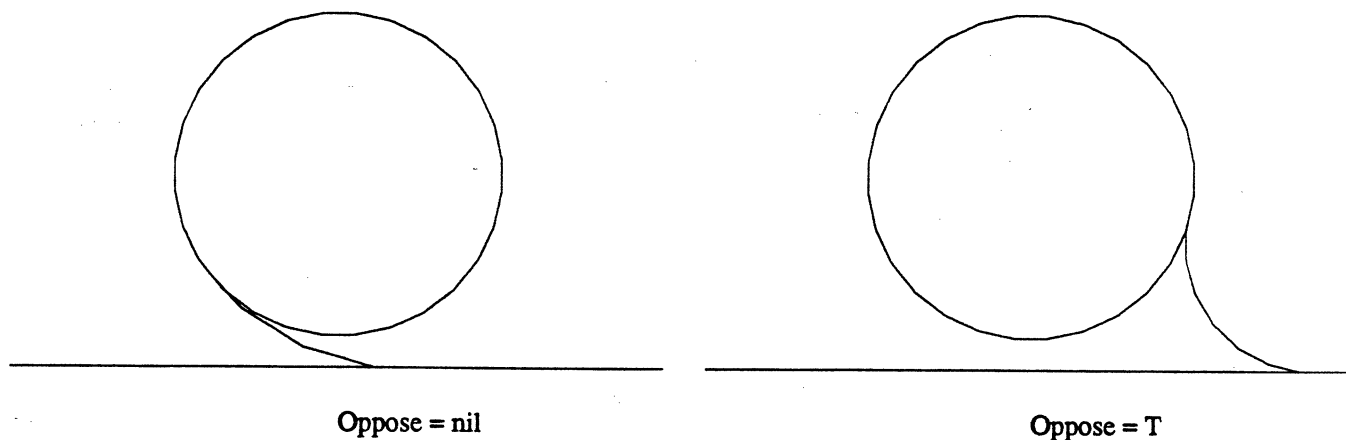


Figure 7-5: Oppose Flag for ArcRadTanToCircleAndLine

a circle and a line. Three flags, together with the direction of the given line, specify which of the possible arcs will be constructed.

If the *Oppose* flag is true, the arc will be “opposing” the given circle. An opposing arc always lies outside the circle. A non-opposing arc may lie outside the circle if the tangent line doesn’t intersect the circle. Another way to think about the opposing arcs: if you travel the given circle and the desired arc clockwise, do the directions of travel agree or disagree at the point of tangency? If they disagree, the arc opposes the circle. Figure 7-5 shows the two arcs generated with different values of the *Oppose* flag.

If the *Side* flag is true, the resulting arc is on the same side of the tangency line as the center of the circle. For cases where the line does not intersect the circle, *Side* must be given as true.

The *ArcOrient* flag does not affect the particular arc which is generated, but only its resulting orientation. If *ArcOrient* is Nil, then any arc generated starts on the circle and ends on the line. If you need the arc to start on the line and end on the circle, then *ArcOrient* should be set to true (T).

Note that the orientation of the tangent line does affect which arc is chosen. In the normal case where *ArcOrient* is Nil, and the direction of the arc proceeds from the circle to the line, the direction of the line and the arc will agree at the point of tangency. Depending on which arc you are trying to generate, certain radius values for the desired arc will not be valid. The routine checks for these at the beginning and informs you if the radius is too small or too large.

**arcRadTanToCircleAndLine( Rad, Cir, Tan, Oppose, Side, ArcOrient )**

<b>Returns</b>	<arc> Construct an arc with a given radius tangent to a circle and a line.
<b>Rad</b>	<number> The radius of the arc.
<b>Cir</b>	<circle> The circle to to be tangent to.
<b>Tan</b>	<line> The line to be tangent to.
<b>Oppose</b>	<opt boolean> If true, create an “opposing” arc (which always lies outside the circle), the arc and circle will have opposite directions at the point of tangency.

**Side** <opt boolean> If true, the resulting arc is on the same side of the tangent line as the center of the circle.

**ArcOrient** <opt boolean> If true, the arc will start on the line and end on the arc.

### Arc Extractors

Operations are provided to extract the radius or center of an arc from a constructed arc (**radiusOfArc**, **centerOfArc**) and to access the first and last points of an arc (**arcStart**, **arcEnd**).

**radiusOfArc**( Arc )

**Returns** <number> Determine the radius of a given arc.

**Arc** <arc> The arc to examine.

**centerOfArc**( Arc )

**Returns** <e3Pt> Determine the center of a given arc.

**Arc** <arc> The arc to examine.

**arcStart**( Arc )

**Returns** <e3Pt> Determine the first point of an arc.

**Arc** <arc> The arc to examine.

**arcEnd**( Arc )

**Returns** <e3Pt> Determine the last point of an arc.

**Arc** <arc> The arc to examine.

**startDirOfArc**( Ar )

**Returns** <geomVector> A vector given the tangent direction at the beginning of the arc.

**Ar** <arc> The arc to examine.

**endDirOfArc**( Ar )

**Returns** <geomVector> A vector given the tangent direction at the end of the arc.

**Ar** <arc> The arc to examine.

**directionOfArc**( Ar )

**Returns** <keyword> Returns 'CW or 'CCW indicating the direction of the arc.

**Ar** <arc> The arc to examine.

An arc is part of a circle. To construct an arc representing the other part of the circle, use **otherArc**. The result is an arc in the other direction with the same starting and ending points. Unlike most of the other arc constructors, **otherArc** can generate arcs subtending any angle less than 360 degrees.

**otherArc**( Arc )

**Returns** <arc> Construct the missing portion of the given arc.

**Arc** <arc> The arc to complete.

### Circles

A limited number of circle operations are available. By default, a circle lies in a plane parallel to the XY plane, presumably lying in that plane. Several routines are provided for constructing circles. The simplest ones require the center and radius (**circleCtrRad**) or the center and a point



on the circle (**circleCtrPt**) to be specified. However, circles may be constructed in any 3D position by specifying a normal vector along with the center and radius, using **circleCtrRadNormal**.

**circleCtrRad( Ctr, Rad )**

**Returns** <circle> Construct a circle from a center point and radius.

**Ctr** <euclidPoint> The center of the circle.

**Rad** <number> The radius of the circle.

**circleCtrPt( Ctr, EdgePoint )**

**Returns** <circle> Construct a circle from a center point and a point on its edge.

**Ctr** <euclidPoint> The center of the circle.

**EdgePoint** <euclidPoint> A point on the edge.

**circleCtrRadNormal( Ctr, Rad, Normal )**

**Returns** <circle> Construct a circle from a center point, radius, and normal vector.

**Ctr** <euclidPoint> The center of the circle.

**Rad** <number> The radius of the circle.

**Normal** <geomVec> The normal vector of the plane in which the circle will lie.

To retrieve the center and radius from a constructed circle, use **circleCtr** and **circleRad**.

**circleCtr( Circle )**

**Returns** <euclidPoint> Determine the center point of a circle.

**Circle** <circle> The circle to examine.

**circleRad( Circle )**

**Returns** <number> Determine the radius of a circle.

**Circle** <circle> The circle to examine.

To construct a circle with a given radius, tangent to two other circles, use **circleRadTan2Circles**.

**circleRadTan2Circles( Rad, Cir1, CSW1, Cir2, CSW2 )**

**Returns** <circle> Construct a circle with a given radius which is tangent to two other circles.

**Rad** <number> The radius of the circle.

**Cir1, Cir2** <circle> The two target circles to be tangent to.

**CSW1, CSW2**

<boolean> If true, the resulting curve should arc the same way as the target curve at the point of tangency.

Two flags are given along with the circles, indicating whether the new circle will curve the same way (T) or the opposite way (Nil) at the tangency points with the given circles. Usually, there are two circles which could be constructed. The one which is returned is the one whose center would be to the right of a line from the center of the first circle given to the center of the second one. The center of the new circle is found by intersecting a pair of offset circles whose radii are increased by the radius of the new circle over the radius of the given circle if the circles curve in opposite ways at the tangency point, and offset to a smaller radius if the circles curve the same way. (Negative radii mean the offset circle radius has passed through zero and started to grow again.) There are an amazing number of combinations of radii, circle positions, and CurveSameWay flags which construct proper circles. Invalid combinations will cause **ptIntersect2Circles** to report an error because the offset circles don't intersect.

The circle which results from **circleRadTan2Circles** can be trimmed down to just the arc between the tangency points of the circles using **arcRadTan2Circles**.

**arcRadTan2Circles( Rad, Cir1, CSW1, Cir2, CSW2 )**

**Returns** <arc> Construct an arc with the given radius which is tangent to two circles.

**Rad** <number> The radius of the resulting arc.

**Cir1, Cir2** <circle> The two target circles the arc will be tangent to.

**CSW1, CSW2**

<boolean> If true, the arc will curve the same way as the circle at the point of tangency.

An arc can be "cut" from a circle with a line using **arcCutFromCircle**. The direction of the resulting arc is the same as that of the line. The intersection of a line with a circle can be calculated with **ptIntersectCircleLine**. The resulting point is at the first intersection of the circle and the line. Reversing the line would give the other intersection point. An error occurs if the circle and line do not intersect, but the case where they are tangent is handled correctly. Two circles intersect (in general) at two points. One of the intersections is calculated with **ptIntersect2Circles**. The result is the intersection on the right of the line from the center of the first circle to the center of the second circle. (The other one may be had by reversing the order of the circle arguments.) The single intersection point is returned when the circles just come tangent.

**arcCutFromCircle( Cir, Line )**

**Returns** <arc> Cut a circle with a line and return the arc.

**Cir** <circle> The circle to cut.

**Line** <line> The line to cut it with.

**ptIntersectCircleLine( Cir, Line )**

**Returns** <euclidPoint> Construct a point at the intersection of a circle and a line.

**Cir** <circle> The circle to intersect.

**Line** <line> The line to intersect it with.

**ptIntersect2Circles( Cir1, Cir2 )**

**Returns** <euclidPoint> Construct a point at the intersection of two circles.

**Cir1, Cir2** <circle> The two circles to intersect.

The predefined circle, **UnitCircle** is centered on the origin and has radius 1. Note however, that **UnitCircle** is not a circle object, but rather a curve object. The name is confusing, but it is usually more convenient to have the unit circle represented as a curve, rather than as a circle object from which the curve will have to be derived.

**unitCircle**

<curve> A circle centered at the origin with a radius of one.

## 7.9 Basic Curves

This section and the next one describe the basic lowest level constructions which are provided for B-spline curves and surfaces. These constructors allow specification of the defining control points of a curve or surface and the corresponding parametric information to generate an instance of a curve or surface. In general, this will not be the best way to create curves and surfaces with certain

desired shapes. Higher level curve and surface construction operations are introduced in the next chapter.

### Curve Constructors

The lowest-level curve construction operation simply involves specifying the parametric and geometric information which defines the curve. A curve can be constructed by providing a *parmInfo* and a list of defining control points. The construction of the parametric information (*parmInfo*) will be discussed later in this section. The list of defining control points may actually take several forms: a list, a lisp vector, or a polyline.

**curve( *ParmInfo*, *ControlPts* )**

**Returns**     <curve> Construct a B-spline curve from a set of control points.

**ParmInfo**    <parmInfo> The parametric information for the curve.

**ControlPts**   <ctlPoly | polyline | ctlMeshRow | listOf point> The set of points to use as the control polygon.

There are many other ways to create B-spline curves. One is to construct an arc and convert it into a B-spline curve. Note that arcs and curves are different geometric entities in Alpha\_1, although some routines which work with curves will accept arcs and convert them to curves automatically. The **crvFromArc** and **crvFromCircle** operations are used for converting arcs and circles to B-spline curves.

**crvFromArc( *Arc* )**

**Returns**     <curve> Coerce an arc to a B-spline curve.

**Arc**           <arc> The arc to convert.

**crvFromCircle( *Circ* )**

**Returns**     <curve> Coerce a circle into a curve.

**Circ**           <circle> The circle to convert.

It is very often useful to chain together sequences of points, arcs, and curves to form a single curve. The **profile** operation chains together curves, arcs, and points (which may have shared endpoints), making a composite curve. A line segment is used to connect the endpoints of adjacent pieces if necessary. The resulting B-spline curve contains all the corners in the order listed, with each neighboring pair of endpoints connected by a straight line segment. Currently, the curves to be joined must have end condition type **EC\_OPEN**, but there are no restrictions on the order. The resulting curve will have the highest order of the input curves, with lower order curves converted to that order before chaining them together. Any duplicate points are dropped and the knot vectors are joined with knots of multiplicity Order-1 where there were two knots of multiplicity Order originally. This implies a piecewise Bezier type of curve, with no continuity except position guaranteed at the places where the curves were joined.

**profile( *Corner1*, ... )**

**Returns**     <curve> Concatenate a sequence of items into a composite curve.

**Corner1, ...**   <euclidPoint | arc | curve> An arbitrary sequence of points, arcs, and curves to concatenate.

Note that an easy way to create a linear (straight line) curve between two points is with

**StraightCurve := profile( Pt1, Pt2 );**

As usual, the direction of a curve (which end point is listed first) may be inverted using `reverseObj`. As an example of the curve construction operators, consider describing the rounded "X" shape in Figure 7-6.

One construction sequence might involve defining the curve in the upper right quadrant and reflecting it to achieve the entire figure:

```
% Define the two 45-degree lines.
Lower45Line ^= linePtAngle( pt( 0.1, 0.0 ), 45 );
Upper45Line ^= linePtAngle( pt( 0.0, 0.1 ), -135 );

BaseOffset ^= 0.2;      % Offset from origin for end of arc.
BaseRadius ^= 0.2;      % Radius of the arc joining the arms.

% Construct the two arcs at the base of the arm.
RightArc ^= arcRadTan2Lines( BaseRadius,
                             lineVertical BaseOffset,
                             Lower45Line );
LeftArc ^= arcRadTan2Lines( BaseRadius,
                             Upper45Line,
                             reverseLine lineHorizontal BaseOffset );

% Construct the 180-degree arc at the end of the arm.
EndLine ^= linePtAngle( pt( 1, 0 ), 135 );
EndArc ^= arcTan3Lines( Lower45Line, EndLine, Upper45Line );

% Put all three arcs together into one curve.
QuarterCurve ^= profile( RightArc, EndArc, LeftArc );

% Perform reflections to build the complete curve.
TopHalf ^= profile( QuarterCurve,
                   reverseObj reflect( QuarterCurve, 'x' ) );
BotHalf ^= reflect( TopHalf, 'y' );

% Connect top and bottom half. Final point causes profile to
% close the curve.
FinalCurve ^= profile( TopHalf,
                      reverseCrv BotHalf,
                      topHalf->cPoly[0] );
```

### Parametric Information

As mentioned at the beginning of this section, B-spline curves and surfaces are described in two parts: the parametric information, and the geometric information. Parametric information includes the polynomial order of the curve or surface, the end condition type, and the knot vector. Geometric information is just the defining control points. The parametric and geometric information together describe a curve or surface. Of the constructors for curves which were discussed above, only the lowest-level "curve" operator requires explicit specification of the parametric information. The other constructors produce an appropriate parametrization which depends on the given data and the constructor.

The structure which contains the parametric information for a B-spline curve is called a *parmInfo* in Alpha\_1. A *parmInfo* may be constructed by invoking the operator of the same name:

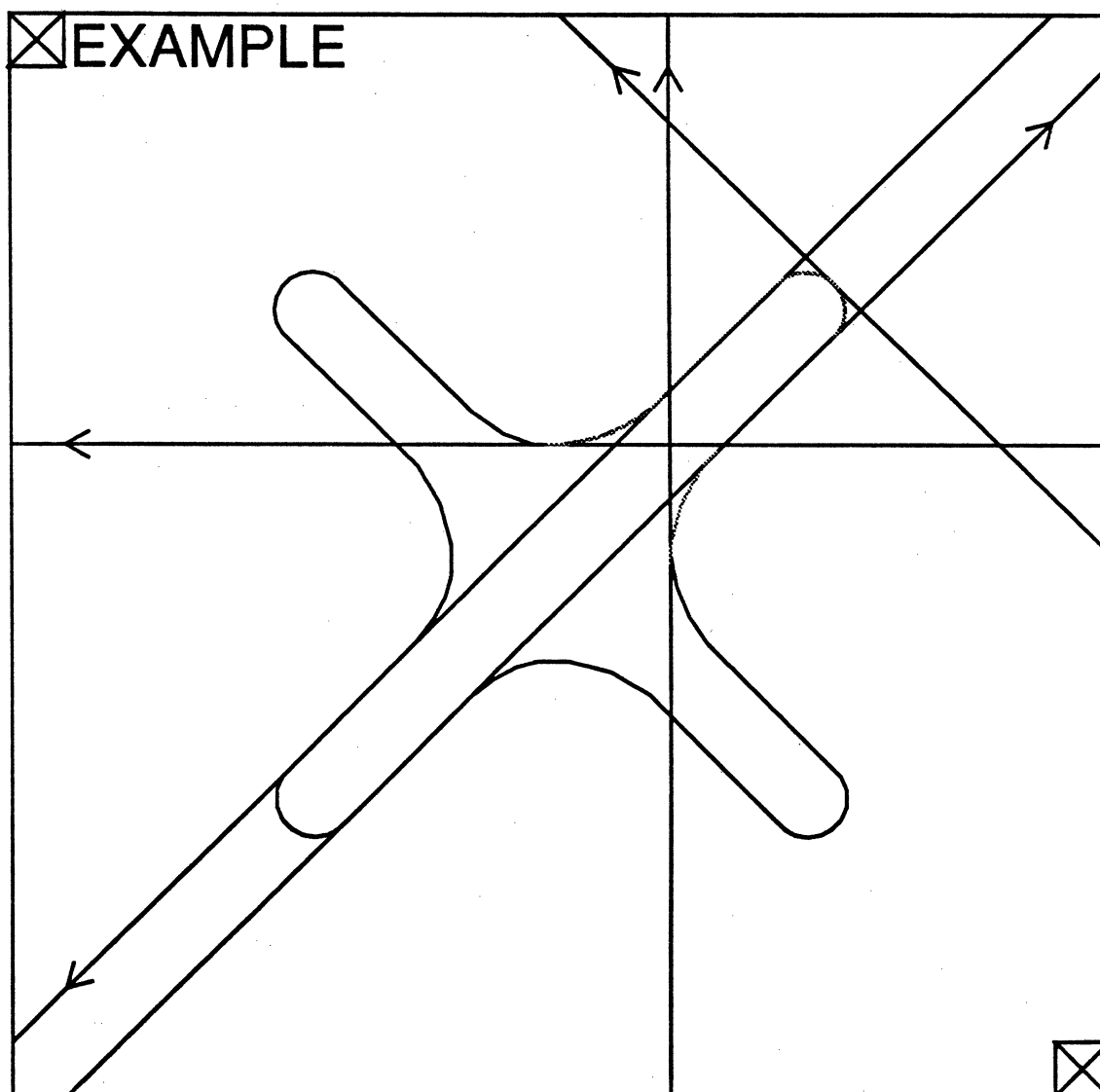


Figure 7-6: Example of 2D Curve Construction

**parmInfo**( *Order*, *EndCond*, *KnotVec* )

**Returns**     <parmInfo> Create a parametric information structure for constructing B-spline curves.

**Order**        <integer | symbol> The polynomial order of the curve. This can be given as an integer, or as one of the symbols **LINEAR**, **QUADRATIC**, **CUBIC**, **QUARTIC**, **QUINTIC**.

**EndCond**     <symbol> The end condition type of the curve. This is one of the symbols **EC\_OPEN**, **EC\_FLOATING**, or **EC\_PERIODIC**.

**KnotVec**     <knotVector | listOf number | vectorOf number | symbol> The knot vector of the curve. The knot vector may be given explicitly, as a list of non-decreasing floating point values, or it may be specified as one of the symbols **KV\_UNIFORM**, **KV\_BEZIER**, **KV\_CHORD**. Those special knot vector specifications have the following interpretations:

**KV\_UNIFORM**

Interior knots are single and evenly spaced. In general, if there is no special reason to use the other kv types, this is the one you should choose.

**KV\_BEZIER**

The knot vector has interior knots of multiplicity Order-1. This causes the curve to behave as if it were a piecewise Bezier curve.

**KV\_CHORD**

Interior knots are single as for the uniform case, but are spaced proportionally to the chord length between control polygon vertices.

There are several extraction operators for requesting particular pieces of information about a parmInfo. The order, **parmOrder**, is a small integer. The end condition type, **parmEndCondType**, is one of the symbols **EC\_OPEN**, **EC\_FLOATING**, or **EC\_PERIODIC**. The type of the knot vector, **parmKvType**, is Nil if the knot vector was given explicitly. Otherwise, it is one of **KV\_BEZIER**, **KV\_UNIFORM**, or **KV\_CHORD**. The value of the knot vector, **parmKvValue**, is Nil unless the knot vector was given explicitly. In this case, it will be the vector of knot values. For knot vectors specified by one of the predefined knot vector types, the value of the knot vector will remain Nil until the parmInfo is associated with a spline curve or surface. (The actual values in the knot vector depend on the number of control points.)

**parmOrder**( *ParmInfo* )

**Returns**     <integer> Determine the order declared by a parmInfo object.

**ParmInfo**    <parmInfo> The parmInfo structure to examine.

**parmEndCondType**( *ParmInfo* )

**Returns**     <symbol> Determine the end condition type declared in a parmInfo object.

**ParmInfo**    <parmInfo> The parmInfo structure to examine.

**parmKvType**( *ParmInfo* )

**Returns**     <symbol> Determine the knot vector type of the parmInfo. This is Nil if the knot vector was given explicitly.

**ParmInfo**    <parmInfo> The parmInfo structure to examine.

**parmKvValue**( *ParmInfo* )

**Returns** <KnotVector> Get the knot vector defined by a parmInfo structure.  
**ParmInfo** <parmInfo> The parmInfo structure to examine.

Some of the operators for curves and surfaces require that the curves be of end condition type **EC\_OPEN**, and so it may be necessary to convert a curve with **EC\_PERIODIC** or **EC\_FLOATING** end conditions to **EC\_OPEN** using **curveOpen**. The resulting curve is different only in its representation, not in the geometric space curve which it describes.

**curveOpen( Curve )**

**Returns** <curve> Convert a periodic or floating curve to **EC\_OPEN**.  
**Curve** <curve> The curve to convert.

The lowest-level curve refinement operation is called **cRefine**. The original curve is specified, together with the refinement knot vector. The refinement knot vector may consist of just the new knots which are to be added, in which case **Merge** should be specified as **T**. The refinement knot vector may already contain both the original knots and the new knots to be added, in which case **Merge** should be **Nil**. See chapter 3 [Spline Introduction], page 11 for a description of refinement.

**cRefine( OldCrv, NewKnots, Merge )**

**Returns** <curve> Refine a given curve with a new knot vector. This results in a curve with more flexibility.  
**OldCrv** <curve> The original curve to refine.  
**NewKnots** <knotVector | listOf number | vectorOf number> The new knot vector for the curve.  
**Merge** <boolean> If true, the **NewKnots** argument contains only the new knots and should be merged with the original knot vector.

Raising the order of a curve or surface after it has been created is often a useful operation. This is one way to add extra degrees of freedom for later operations on the curve (although refinement is usually the preferred method). One important case where raising the order is often necessary is for linear splines, which are going to be modified to make smooth, but not straight curves.

The **raiseCurve** function produces a new curve which is geometrically identical to the original, but which is represented as a B-spline of the specified order. If **NewOrder** is less than or equal to the original order of the curve, the original curve is just returned. Curves may often be part of larger aggregate structures such as lists, lisp vectors, groups, shells, instances, and parametric objects. The **raiseOrder** operation raises all the curves and surfaces (both directions) in the object to the given order. If just the curves in a higher level structure need to be changed, use **raiseCrvOrder**. To raise the order of all the curves in a set to a common order, one can use **sameOrder** which returns a list of the resulting curves, which all have order equal to the maximum order of any of the original curves. Many of the operators which require that the curves be the same order will perform this operation automatically as part of the calculations (e.g., **crvConcat** or **boolSum**).

**raiseCurve( Curve, NewOrder )**

**Returns** <curve> Raise the degree of a given B-spline curve.  
**Curve** <curve> The curve to degree raise.  
**NewOrder** <integer> The desired order for the curve.

**raiseOrder( Obj, Order )**

**Returns** <object> Walk through the structure of an object degree raising all the curves and surfaces it contains.  
**Obj** <object> The object to degree raise.

**Order**      <integer> The desired order of the curves.

**raiseCrvOrder( Obj, Order )**

**Returns**    <object> Walk through a structure of an object degree raising only the curves.

**Obj**          <object> The object to degree raise.

**Order**       <integer> The desired order of the curves.

**sameOrder( Crv1, ... )**

**Returns**    <listOf curve> Given a set of curves ensure that all curves have the same order.

**Crv1, ...**    <curve> A set of curves to check.

A set of curves can also be modified so that they share a common knot vector as well as a common order. The **mkCompatible** operator raises the order of the curves as necessary and refines the knot vectors as necessary to produce a list of curves with identical orders and knot vectors. If *Coerce* is non-*Nil*, then the resulting list of curves will all have the same "type" of points. Otherwise, no checking of point types is performed. Again, only the representation of the curves changes, not the geometric space curves which they describe. Many of the operators which require compatible curves will perform this operation automatically.

**mkCompatible( Crv1, ..., CrvN, Coerce )**

**Returns**    <listOf curve> Given a set of curves ensure that all curves have the same order and knot vector.

**Crv1, ..., CrvN**

            <curve> The set of curves to modify.

**Coerce**      <boolean> If true, the resulting curves will all have the point type.

### Curve Evaluators

Any particular point on a curve may be determined by specifying the parametric value of interest and performing a curve evaluation with **crvEval**. The point which is the result of evaluating the curve at the specified parameter value is returned.

**crvEval( Crv, Param )**

**Returns**    <euclidPoint> Calculate a point on a spline curve at some parameter value.

**Crv**          <curve> The curve to evaluate.

**Param**       <number> The parameter value at which to evaluate the curve.

B-spline curves may be differentiated. The result of taking one derivative with **diffCrv** is a curve of order one less than the original. The optional *N* argument allows the *N*th derivative of the curve to be computed, but the value of *N* should be less than the order of the curve.

**diffCrv( Crv, N )**

**Returns**    <curve> Differentiate a B-spline curve.

**Crv**          <curve> The curve to differentiate.

**N**            <opt number> The number of times to differentiate (default 1).

To find the value of the derivative at any particular point on the curve, use the **crvEval** operation described above on the resulting derivative curve, or use **crvDerivEval** or **crvNthDerivEval**.

**crvDerivEval( Crv, Param )**



**Returns**    <euclidPoint> Calculate the value of the derivative of a curve at a particular parameter value.  
**Crv**        <curve> The curve to evaluate.  
**Param**      <number> The parameter value to evaluate it at.

**crvNthDerivEval( Crv, N, Param )**

**Returns**    <euclidPoint> Calculate the value of the Nth derivative of a curve at some parameter value.  
**Crv**        <curve> The curve to evaluate.  
**N**          <number> The number of times to differentiate the curve.  
**Param**      <number> The parameter value at which to evaluate the curve.

### Curve Extractors

Several operations are provided for extracting information from a constructed curve. These may also be used in the left hand side of an assignment statement to set or change the corresponding piece of information about the curve.

The full description of the parametric information (**parmInfo**) associated with the curve is accessed with **cParmInfo**. The individual parts of the parametric information associated with a curve can also be accessed directly. The type of the curve returned by **cType** is its end condition type. The **KvType** is the type of the knot vector which was specified on creation of the curve (e.g., **KV\_UNIFORM**). If an explicit knot vector was given the type will be **Nil**. The **cKv** extractor returns the actual numeric values which make up the knot vector. If the knot vector was specified on creation as one of the pre-defined classes of knot vector, it may be necessary to generate the actual values at this time (unless some other operation, like displaying the curve, has already done so). This will be done automatically.

**cParmInfo( Curve )**

**Returns**    <parmInfo> Retrieve the parametric information for a curve.  
**Curve**      <curve> The curve to examine.

**cOrder( Crv )**

**Returns**    <integer> Retrieve the order of a curve.  
**Crv**        <curve> The curve to examine.

**cType( Crv )**

**Returns**    <symbol> Retrieve the end condition type of the curve. This will be one of: **EC\_OPEN**, **EC\_FLOATING**, or **EC\_PERIODIC**.  
**Crv**        <curve> The curve to examine.

**cKvType( Crv )**

**Returns**    <symbol> Retrieve the knot vector type of the curve. This will be one of: **Nil**, **KV\_UNIFORM**, **KV\_CHORD**, or **KV\_BEZIER**.  
**Crv**        <curve> The curve to examine.

**cKv( Crv )**

**Returns**    <knotVector> Retrieve the knot vector of the curve.  
**Crv**        <curve> The curve to examine.

The control polygon extractor, **cPoly**, returns the set of defining control points of the curve.

**cPoly( Crv )**

**Returns** <ctlPoly> Retrieve the control polygon of a curve.  
**Crv** <curve> The curve to examine.

A single control point of a curve can be retrieved with **crvPt**, or all the control points retrieved in a list with **ptListFromCtlPoly**.

**crvPt( Crv, Indx )**

**Returns** <point> Extract a control point of the curve.  
**Crv** <curve> The curve to be examined.  
**Indx** <integer> Index of the desired control point.

**ptListFromCtlPoly( CtlPoly )**

**Returns** <listOf point> A list of the points of the control polygon.  
**CtlPoly** <ctlPoly> The control polygon.

Another operation, **regionFromCrv**, allows extraction of a region of a curve between two specified parametric values.

**regionFromCrv( Crv, LowParamVal, HighParamVal )**

**Returns** <curve> Extract a curve from a given curve which is bounded by two parametric values.  
**Crv** <curve> The reference curve.  
**LowParamVal, HighParamVal** <number> These are the beginning and ending parametric values of the desired curve.

Finally, the nodes of the curve (or a parmInfo) can be computed with **computeNodes**.

**computeNodes( KnotInfo )**

**Returns** <vectorOf number> A vector containing the nodes corresponding to the knot vector.  
**KnotInfo** <curve | parmInfo> A curve or parmInfo for which nodes are to be computed.

## 7.10 Basic Surfaces

The basic low-level surface construction operation is called **surface**.

**surface( ParmInfos, Mesh )**

**Returns** <surface> Construct a surface from parametric information and a control mesh.  
**ParmInfos** <Nil | parmInfo | listOf parmInfo> The parametric information structure, it can be Nil, a single parmInfo (which will be used for both U and V parametric directions), or a list of two parmInfos.  
**Mesh** <ctlMesh | vectorOf point | listOf point> The control mesh for the surface. It must have control polygons for each parametric direction.

For B-spline curves, a single parmInfo gives all the parametric information which is needed to define the curve. The B-spline surfaces however, are bivariate functions, which means that a pair of

parmInfos must be specified, one for each parametric direction. These two parmInfos are completely independent. They may have different orders, different end conditions, and different knot vectors. One of these parmInfos corresponds to the row direction of the control mesh, the other to the column direction. We usually use the symbols **ROW** and **COL** as indexes and as arguments when an operation requires a direction to be specified. Generally, two sets of parametric information are specified for a surface. However, if only one is given it is duplicated and used for both parametric directions in the resulting surface. The *control mesh* is the array of defining control points for the surface. Each row of the control mesh is a list or vector of points, and the control mesh is a list or vector of these rows. There are many other surface constructors in the system which provide much less tedious ways of defining surfaces than specifying all the control points.

Surfaces, like curves, can be refined by adding knots to produce extra degrees of freedom. Once again, this operation changes only the representation of the surface, and not its geometric shape. For surfaces, only one parametric direction at a time is refined. That is, the refinement operation adds extra knots to all the rows of the surface, or to all the columns of the surface. The surface refinement operator is **sRefine**. The original surface is provided, with the direction (either **ROW** or **COL**) which is to be refined. The new knot vector may be either just the new knots to be added (in which case *Merge* should be **T**), or the combined set of the original knots with the new knots (in which case *Merge* should be **Nil**).

**sRefine**( *OldSrf*, *Dir*, *NewKnots*, *Merge* )

*Returns*     <surface> Refine a surface in a given direction using the given knot vector.  
*OldSrf*     <surface> The original surface to refine.  
*Dir*         <symbol> The direction to refine in, one of the symbols **ROW** or **COL**.  
*NewKnots* <knotVector | listOf number | vectorOf number> The new knot vector to use.  
*Merge*      <boolean> If true, the new knot vector contains only new knots and should be merged with the original knot vector.

A surface may be converted from end condition types of **EC\_PERIODIC** or **EC\_FLOATING** to **EC\_OPEN** using **srfOpen**. This is necessary for some operators which require open end conditions for their surface arguments.

**surfaceOpen**( *Srf* )

*Returns*     <surface> Convert a surface to have open end conditions.  
*Srf*          <surface> The surface to convert.

The orientation of a surface is implicit in the ordering of the control mesh, as shown in Figure 7-7, see chapter 5 [Orientation Conventions], page 29 for more on this.

The rows of the control mesh extend in the *U* parametric direction while the columns extend in the *V* parametric direction. The surface normal points into the page. The orientation of a surface may be reversed using the **reverseObj** operation. The **reverseObj** operation transposes the control mesh (and exchanges the knot vectors) to reverse the orientation of a surface. Sometimes it is necessary to reverse a surface by flipping it end-for-end in either the column or the row direction with **reverseSrfInDir**. The isoparametric curves in the specified direction (**ROW** or **COL**) are reversed.

**reverseSrfInDir**( *Srf*, *Dir* )

*Returns*     <surface> Reverse a surface in a certain direction by flipping it end-for-end.  
*Srf*         <surface> The surface to flip.  
*Dir*         <symbol> The direction to flip, one of **ROW** or **COL**.

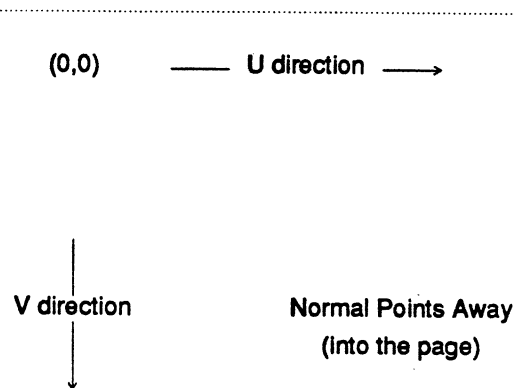


Figure 7-7: Orientation of Surfaces

### Surface Extractors

The extraction operators for surfaces are similar to those for curves. They may be used as usual on the left hand side of an assignment statement to set the value of the corresponding information, as well as to extract the information from a surface.

The complete parametric information (both of the `parmInfos`) is accessed using `sParmInfos`. The first part of the resulting vector is the row (or U-parameter) information and the second part is the column (or V-parameter) information. The two parts may also be accessed using `rowParmInfo` or `colParmInfo`. Individual parametric information fields can also be accessed directly. In each of the operations below, a vector of two values is returned. The first corresponds to the rows (U parameter) and the second to the columns (V parameter) of the surface. The surface types (`sTypes`) are the end condition types, while the `kvTypes` are the knot vector types (if they have been specified). The knot vectors returned by `sKvs` are the actual vectors of knot values which may have been specified initially on creation of the surface or generated based on the `KvTypes` from the other surface information.

**sParmInfos( Surface )**

**Returns**    <vectorOf parmInfo> Retrieve the parametric information from a surface, `parmInfos` for both directions are returned.

**Surface**    <surface> The surface to examine.

**rowParmInfo( Srf )**

**Returns**    <parmInfo> The parametric information in the row direction.

**Srf**            <surface> The surface to examine.

**colParmInfo( Srf )**

**Returns**    <parmInfo> The parametric information in the column direction.

**Srf**            <surface> The surface to examine.

**sOrders( Srf )**

**Returns**    <vectorOf integer> Retrieve the U and V orders of a surface.

**Srf**            <surface> The surface to examine.

**sTypes( Srf )**

**Returns**    <vectorOf symbol> Retrieve the end condition types of a surface.

**Srf**            <surface> The surface to examine.

**sKvTypes( Srf )**

**Returns**    <vectorOf symbol> Retrieve the knot vector types of a surface if they are one of KV\_UNIFORM, KV\_CHORD, KV\_BEZIER, otherwise return Nil.

**Srf**            <surface> The surface to examine.

**sKvs( Srf )**

**Returns**    <vectorOf knotVector> Retrieve the knot vectors for a surface.

**Srf**            <surface> The surface to examine.

To extract the control mesh, use **sMesh**.

**sMesh( Srf )**

**Returns**    <ctlMesh> Retrieve the control mesh used by a surface.

**Srf**            <surface> The surface to examine.

Another extraction operator, **crvFromSrf**, constructs a B-spline curve from one of the rows or columns of the surface control mesh. The parametric information corresponding to the row or column direction of the surface is duplicated for the resulting curve. The index indicates which row or column is desired (counting from 0). This extractor may not be used on the left hand side of an assignment statement as the others could. Note that the resulting B-spline curve does not necessarily lie in the surface from which it was extracted. The surface is a blend of the "curves" defined by the rows or columns of the control mesh. The result of **crvFromSrf** is just the curve which happens to be defined by a particular row or column of the control mesh. A common use of **crvFromSrf** is to extract the boundary curves of a surface, so a special operator, **getBoundary** is provided for this case. It returns the boundary curve of the surface along the given edge ('TOP', 'BOTTOM', 'LEFT', or 'RIGHT'). Top and bottom edges correspond to the first and last rows of the control mesh respectively, and left and right correspond to the first and last columns respectively.

**crvFromSrf( Srf, Dir, Index )**

**Returns**    <curve> Extract a curve corresponding to a particular row or column from a surface given a direction and a row or column index.

**Srf**            <surface> The surface to examine.

**Dir**            <symbol> One of ROW or COL.

**Index**        <integer> The row or column index of the desired curve. For convenience the symbol **LAST** may be used to refer to the largest index in the chosen direction.

**getBoundary( Srf, Edge )**

**Returns** <curve> Extract the desired boundary curve from the surface.  
**Srf** <surface> The surface to examine.  
**Edge** <symbol> The desired boundary curve, must be one of 'TOP', 'BOTTOM', 'LEFT', or 'RIGHT'.

To extract a particular (isoparametric) curve which lies in the surface, use **crvInSrf**. This operation returns an isoparametric curve in the U (row) or V (column) direction at the given parametric value. A region of the surface may be extracted using **regionFromSrf**. The operation returns a surface which is the region from *LowParamVal* to *HighParamVal* in the given direction.

**crvInSrf( Srf, Dir, ParamVal )**

**Returns** <curve> Extract a curve corresponding to a particular parametric value from a surface.  
**Srf** <surface> The surface to examine.  
**Dir** <symbol> One of ROW or COL.  
**ParamVal** <number> The parametric value of the desired curve.

**regionFromSrf( Srf, Dir, LowParamVal, HighParamVal )**

**Returns** <surface> Extract a new surface by "slicing" an existing surface.  
**Srf** <surface> The surface to slice.  
**Dir** <symbol> One of ROW or COL.  
**LowParamVal, HighParamVal** <number> The parametric values which will be the boundaries in the indicated direction of the new surface.

Like curves, it is often useful to be able to raise the order of a surface after it is created to provide more degrees of freedom for later shaping operations. Surfaces have two orders which may be raised, one in each parametric direction. The **raiseSurface** function will raise the surface to the given order in the specified direction. The *Dir* argument should be ROW, COL, or Nil. Nil specifies that the order is to be raised in both directions. The more general **raiseOrder** raises the order of all curves and surfaces contained in a structure. Surfaces are raised in both parametric directions. If only surfaces in the object need to be raised, use **raiseSrfOrder**.

**raiseSurface( Srf, Dir, Order )**

**Returns** <surface> Raise the order of a surface.  
**Srf** <surface> The surface to raise.  
**Dir** <symbol | Nil> The direction to be raised, if Nil is given raise both directions.  
**Order** <integer> The new order for the surface.

**raiseOrder( Obj, Order )**

**Returns** <object> Walk an object raising the order of all curves and surfaces in it.  
**Obj** <object> The object to walk.  
**Order** <integer> The new order of the object.

**raiseSrfOrder( Obj, Dir, Order )**

**Returns** <object> Walk an object raising the order of all surfaces (only) in it.  
**Obj** <object> The object to walk.

*Dir*            <symbol> One of **ROW** or **COL**.  
*Order*        <integer> The new order for the object.

The **srfMerge** operation allows two surfaces to be merge into one. The geometry can be allowed to change slightly at the boundary to guarantee some continuity between them, or they can be declared to be an exact match.

**srfMerge**( *Srf1*, *Srf2*, *Dir*, *JoinType* )

*Returns*      <surface> A single surface merged from the original pair.  
*Srf1*, *Srf2*    <surface> The surfaces to be merged together.  
*Dir*            <symbol> The direction in which to perform the merge. If rows are specified, the rows of the meshes will be concatenated.  
*JoinType*     <keyword> Can be **'EXACT'** or **'BLEND'**, specifying how the region at the join is to be treated. If an exact join is specified, the surfaces are assumed to share a boundary curve, and the control points along one of the surface edges are dropped.

The **srfFromCrvs** and **srfFromCrvsDir** functions approximate a set of spline curves with a surface by using each curve control polygon as a row or column of a surface control mesh.

**srfFromCrvs**( *ParmInfo*, *Crv1*, ... )

*Returns*      <surface> An approximating surface for the set of curves.  
*ParmInfo*     <parmInfo> Parametric information for the new surface direction. (The other direction will be absorbed from the curve parmInfos.)  
*Crv1*         <curve> The set of curves to approximate.

**srfFromCrvsDir**( *ParmInfo*, *Dir*, *Crv1*, ... )

*Returns*      <surface> An approximating surface for the set of curves.  
*ParmInfo*     <parmInfo> Parametric information for the new surface direction. (The other direction will be absorbed from the curve parmInfos.)  
*Dir*           <symbol> The direction (row or column) in which the curves are to be placed.  
*Crv1*         <curve> The set of curves to approximate.

### Surface Evaluation Operations

Occasionally it is necessary to evaluate points on a surface. This can be done with **srfEval**. The derivative a surface is computed with **diffSrf**, and surface subdivision can be performed with **srfDivide** or **srfSubdiv**. The **surfaceNormals** function computes a set of surface normal vectors.

**srfEval**( *Srf*, *U*, *V* )

*Returns*      <point | number | vector> Evaluation of the surface at parameter (U,V).  
*Srf*            <surface> The surface to be evaluated.  
*U*, *V*         <number> Parametric coordinates of the point to be evaluated.

**diffSrf**( *Srf*, *UNum*, *VNum* )

*Returns*      <surface> Differentiate the surface the specified number of times in each direction.  
*Srf*            <surface> The surface to be differentiated.  
*UNum*, *VNum*   <number> The number of times to differentiate in each direction.

**srfDivide( Srf, Dir )**

**Returns** <listOf surface> Two subdivided pieces of the surface.

**Srf** <surface> The surface to be subdivided.

**Dir** <symbol> Direction in which to subdivide.

**srfSubdiv( Srf, Dir, MidIndex, SplitMul, KnotVal )**

**Returns** <listOf surface> Two subdivided pieces of the surface.

**Srf** <surface> The surface to be subdivided.

**Dir** <symbol> Direction in which to subdivide.

**MidIndex** <integer> Index of the knot next to the subdivision point.

**SplitMul** <integer> Multiplicity of the knot to be added.

**KnotVal** <number> Parametric value at which to subdivide.

**surfaceNormals( Srf, UVals, VVals )**

**Returns** <vectorOf vectorOf r3Vec> A table of normals at the Cartesian product of the specified U and V values.

**Srf** <surface> The surface for which normals are to be computed.

**UVals, VVals**

<listOf number> Parametric values at which to evaluate in each direction.

It is possible to calculate some differential geometry values for surface. The **srfGeom** routine returns an object which contains the position, parametric derivatives, unit normal, gaussian curvature, mean curvature, principle curvatures, and principle directions for a surface at a given parametric value.

**srfGeom( Srf, U, V )**

**Returns** <srfGeom> Computes differential geometry values for a point on the surface.

**Srf** <surface | srfAndDerivs> The surface to be examined (or a special structure with pre-computed derivatives, described below).

**U** <number> The U parameter value of the point of interest.

**V** <number> The V parameter value of the point of interest.

Using the results of **srfGeom**, a pair of osculating circles or the osculating circle in a principle direction can be calculated.

**prinCircles( SrfGeom )**

**Returns** <group> A group of two osculating circles.

**SrfGeom** <srfGeom> Surface geometry computed with the **srfGeom** function.

**onePrinCircle( Position, Normal, PrinDir, PrinCurv )**

**Returns** <instance> An instance of the unit circle or the y axis, transformed to represent the osculating circle.

**Position** <euclidPoint> The point on the surface.

**Normal** <geomVector> The normal to the surface.

**PrinDir** <geomVector> The principal direction.

**PrinCurv** <number> The principal curvature.

If a number of calculations are going to be made for a single surface, it is much more efficient to pre-compute all the derivatives. The special structure which is returned by **srfAndDerivs** can be



passed on to `srfGeom` in place of the surface itself.

**`srfAndDerivs( Srf )`**

*Returns*    `<srfAndDerivs>` A special structure containing the surface and its derivatives.

*Srf*        `<surface>` The surface for which derivatives are to be computed.

## 8. Curve And Surface Geometry

This chapter describes a set of higher-level operations for constructing curves and surfaces than those presented in the last few sections of the last chapter.

### 8.1 Simple Sweeps

Two traditional ways to form surfaces from curve information are “extrusion” (linear sweeping) and “revolution” (rotational sweeping).

There are two forms of the extrusion operation. The first one, **extrude**, allows the origin of the coordinate system in which the extruded curve is defined to be specified for both ends of the extrusion. Two copies of the curve are made, translated to the new origins in 3D. The two curves form the top ( $V=0$ ) and bottom ( $V=1$ ) parametric edges of the surface which is returned. (That is, the two curves form the rows of the resulting control mesh.) The alternative form of extrusion, **extrudeDir** requires only a vector specifying the direction of the extrusion and (by its length) the extent of the extrusion. The resulting surface is linear in the  $V$  parametric direction with the given curve instanced at  $V=0$ . The corresponding curve at  $V=1$  is the offset of the given curve by the direction vector.

**extrude**( *Crv*, *Pt1*, *Pt2* )

**Returns**     <surface> Construct a solid by extruding a curve given locations for the two end points.

*Crv*             <curve> Curve to be extruded.

*Pt1*, *Pt2*       <point> Instancing origin at each end. The first point will be the  $V = 0$  end, the second will be the  $V = 1$  end.

**extrudeDir**( *Crv*, *Vec* )

**Returns**     <surface> Construct a solid by extruding a curve along a vector.

*Crv*             <curve> Curve to be extruded.

*Vec*             <geomVector> Direction of extrusion. The curve is translated by this vector and connected to *Crv*.

To extrude the rounded “X” shape from the previous example, we could use

```
OutSurface := extrudeDir( FinalCurve, ZDir );
```

The resulting surface is shown with hidden lines removed in Figure 8-1.

Two operations are provided for constructing surfaces of revolution. The simplest surface of revolution, **srfOfRevolution**, rotates a curve about a given axis line. The curve and the axis line must be coplanar. The more general surface of revolution, **srfAxisProfileSection**, allows an arbitrary cross-section shape. For the simple **srfOfRevolution** above, the cross-section is a circle. The cross-section curve is scaled by the distance of each control point of the profile curve from the axis and instanced along the center axis line. To obtain a closed surface, the curve must be closed (either periodic end conditions or open end conditions with the same first and last point).

**srfOfRevolution**( *Axis*, *Profile* )

**Returns**     <surface> Construct a surface by sweeping a profile curve around an axis.

*Axis*           <line> Axis of revolution.

*Profile*        <curve> Curve to be swept about the axis line.

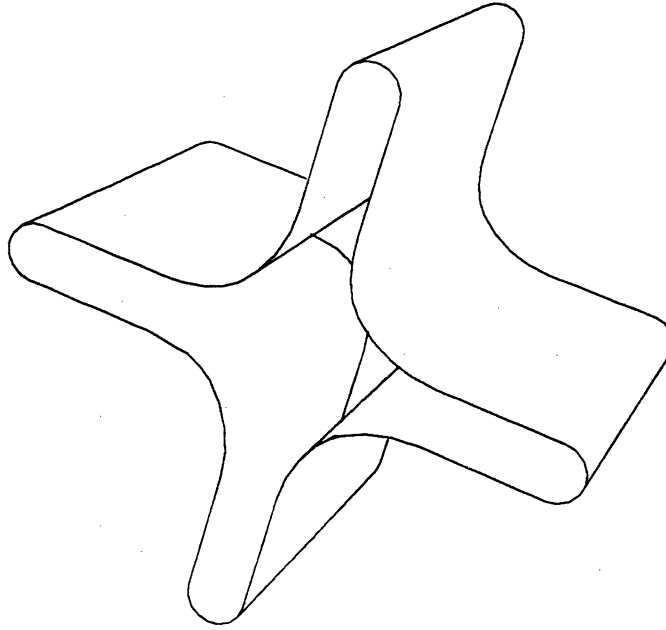


Figure 8-1: Extrusion Operation

**srfAxisProfileSection**( *Axis*, *Profile*, *CrossSect* )

*Returns*     <surface> Construct a surface of revolution using a profile curve, axis line, and an arbitrary cross section.

*Axis*        <line> Axis of revolution.

*Profile*     <curve> Curve to be revolved.

*CrossSect*   <curve> Cross-section shape of result.

Figure 8-2 shows one quadrant from the rounded "X" shape swept about a line at a 135-degree angle:

```
OutSurface := srfOfRevolution( linePtAngle( Origin, 135 ),
                               QuarterCurve );
```

Shells will be discussed later in this manual, but two similar revolution operations, **shellOfRevolution** and **shellAxisProfileSection** are useful for generating closed shell objects. The cross section curve for **shellAxisProfileSection** is quite restricted. It must be a piecewise quadratic Bezier curve with four sections, and it must also be convex. The unit circle (and distortions of it) meet these restrictions. See section 12.1 [Shells], page 181 for more information on shells.

**shellOfRevolution**( *Axis*, *Profile* )

*Returns*     <shell> Construct a solid object from a profile curve swept about an axis.

*Axis*        <line> The axis line about which the profile is swept.

*Profile*     <curve> The profile of the desired solid.

**shellAxisProfileSection**( *Axis*, *Profile*, *CrossSect*, *EndCap* )

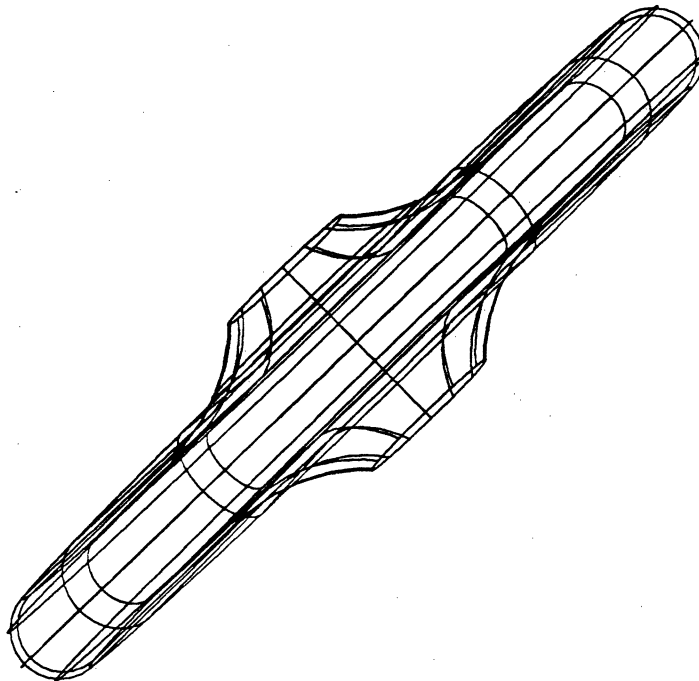


Figure 8-2: Surface of Revolution

*Returns*     <shell> Construct a solid object from a profile curve swept about an axis and following a particular cross section.  
*Axis*         <line> The axis line about which the profile is swept.  
*Profile*       <curve> The profile of the desired solid.  
*CrossSect*   <curve> The cross section outline the profile should follow as it sweeps about the axis.  
*EndCap*      <surface> This is the surface used for each end of the solid.

## 8.2 Volume Primitives

A small set of constructors for basic modeling primitives in the style of the “constructive solid geometry” systems has been provided. These constructors accept traditional parameters for some simple primitives and produce sets of spline surfaces which represent those primitives in the Alpha\_1 style. Sometimes these primitives are a good place to begin to achieve a basic shape which can later be modified with other operators in the system.

### 8.2.1 Basic Primitives

A box is defined by the vertex at one of the corners and three vectors specifying the height, width, and depth of the box relative to the vertex. The box may be arbitrarily oriented. It will have right angles at the corners only if the height, width and depth vectors are mutually perpendicular.

**box**( Vertex, Height, Width, Depth )

**Returns** <box> Construct a parallelepiped given the lengths and orientations of each of its three axes.

**Vertex** <euclidPoint> The reference corner of the box at which the three vectors are based.

**Height, Width, Depth** <geomVector> The dimensions of the box. These can have any orientation and will produce a cube only if mutually perpendicular.

The *geometry slot* of the box, and of all the other primitives described in this section, contains a shell with surfaces that represent the primitive. It can be retrieved from the object using

`objectName->geometry`

if desired. Generally, however, it should not be necessary to examine the geometry field of primitives. The shell structure is how solid volumes are formed from sets of surfaces. See the chapter *Leaving Shape\_edit* for more information on shells. Primitives are examples of parametric types (see chapter 10 [Defining New Object Types], page 153) which allow new types of geometry to be constructed from existing ones. All parametric types contain a geometry field which can be accessed this way.

A right angle wedge is specified by a vertex point and the height, width, and depth vectors relative to the vertex. The wedge has a triangular cross section specified by the height and width vectors, and the vector connecting their endpoints (Figure 8-3). Like the box above, the wedge will only be a right angle wedge if the three vectors are mutually perpendicular.

**rightAngleWedge( Vertex, Height, Width, Depth )**

**Returns** <wedge> Construct a wedge from a reference point and three dimensions.

**Vertex** <euclidPoint> The reference point which serves as the base of the three dimension vectors.

**Height, Width, Depth** <geomVector> The dimensions and orientation for the wedge. The wedge will be a right angle wedge only if the three vectors are mutually perpendicular.

Right circular cylinders, truncated right cones, spheres, tori, and ellipsoids of revolution can also be defined. The base point, height vector, and radius determine a **rightCirCylinder**. A **truncRightCone** is defined by the vertex at the center of the larger base, a height vector, and the radii of the larger and smaller bases respectively. A torus is defined by the vertex at its center, a normal to the plane in which the torus lies (the locus of centers of the circular cross sections lies in this plane), and two scalar radii. The first radius gives the distance from the center to the mid-point of the circular cross section. The second radius is the radius of the circular cross section. An ellipsoid of revolution is defined by the vertex at the center of the axis of revolution, a vector defining the axis of revolution relative to the vertex, and the radius of the circular cross-section at the vertex point. The length of the vector defining the axis of revolution will be the length of the semi-major or semi-minor axis (depending on whether it is larger or smaller than the cross-section radius).

**rightCirCylinder( Vertex, Height, Radius )**

**Returns** <cylinder> Construct a cylindrical solid from a reference point and the cylinder's dimensions.

**Vertex** <euclidPoint> The reference point which serves as the base for the vector and radius.

**Height** <geomVector> The height and axis line of the cylinder.

**Radius** <number> The radius of the cylinder.

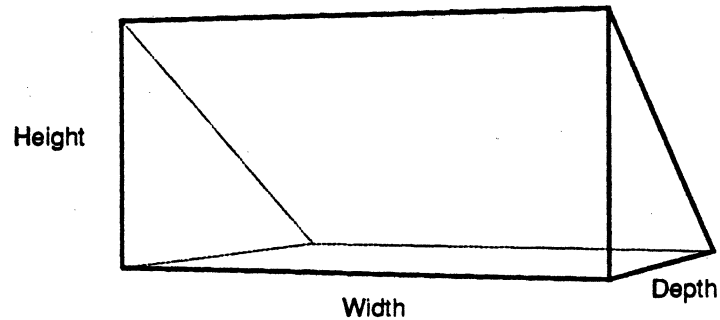


Figure 8-3: Right Angle Wedge Construction

**truncRightCone( Vertex, Height, Bottom, Top )**

**Returns** <cone> Construct a cone from a reference point and dimensions.

**Vertex** <euclidPoint> The reference point for the vector and radii.

**Height** <geomVector> The height and axis line of the cone.

**Bottom, Top**  
<number> The two radii of the cone.

**sphere( Center, Radius )**

**Returns** <sphere> Construct a sphere from a center and radius.

**Center** <euclidPoint> The center of the sphere.

**Radius** <number> The radius of the sphere.

**torus( Vertex, Normal, BigRadius, SmallRadius )**

**Returns** <torus> Construct a torus from a reference point and dimensions.

**Vertex** <euclidPoint> The center of the torus.

**Normal** <geomVector> A vector normal to the plane of the torus.

**BigRadius** <number> The radius of the torus.

**SmallRadius**  
<number> The radius of the cross section of the torus.

**ellipsoid( Vertex, Axis, CrossSectionRadius )**

**Returns** <ellipsoid> Construct an ellipsoid from a reference point, axis and cross section.

**Vertex** <euclidPoint> The reference point which lies on the axis of revolution and is the center of the ellipsoid.

**Axis** <geomVector> The direction defines the axis of revolution. This axis will be the semi-major axis if larger than the *CrossSectionRadius*, and the semi-minor axis otherwise.

**CrossSectionRadius** <number> The radius of the cross section (perpendicular to the *Axis* vector).

## 8.2.2 Rounded Primitives

It is often desirable to have the edges of primitives "rounded" or *filleted* instead of forming sharp angles. A right circular cylinder with rounded top and bottom edges, **roundRightCirCylinder**, is specified just as the normal cylinder. Two extra parameters, *BaseFillet* and *TopFillet*, are also given. They are the radii of the two rounded sections. The first radius is associated with the face of the cylinder in which the vertex point lies. A truncated right cone may also be rounded (**roundTruncCirCone**) by specifying the two additional fillet radii.

**roundRightCirCylinder**( *Vertex*, *Height*, *Radius*, *BaseFillet*, *TopFillet* )

**Returns** <rCylinder> Construct a rounded cylinder from a vertex and dimensions.

**Vertex** <euclidPoint> The reference point which serves as the base for the height and radius.

**Height** <geomVector> The direction and height of the cylinder.

**Radius** <number> The radius of the cylinder.

**BaseFillet, TopFillet** <number> These are the radii of the two rounded edges. The first value is the radius connecting the cylinder wall to the base disk (containing *Vertex*), the second refers to the top disk/wall radius.

**roundTruncCirCone**( *Vertex*, *Height*, *BaseRad*, *TopRad*, *BaseFillet*, *TopFillet* )

**Returns** <rCone> Construct a rounded truncated cone from a vertex and dimensions.

**Vertex** <euclidPoint> The reference point which serves as the base for the height and radius.

**Height** <geomVector> The direction and height of the cone.

**BaseRad, TopRad** <number> The radii of the base and top of the truncated cone.

**BaseFillet, TopFillet** <number> The radii of the rounded edge connecting the base to the cone wall and the top to the cone wall.

The rounded edge box primitive, allows construction of a six-sided box with rounded edges. Specification of the box is accomplished by supplying the plane representations of the faces and associating radii for rounding with the implied edges.

A convention for referring to the sides of the rounded edge box is required for two reasons. The first is to allow reference to the edges for assigning radii and to the faces to specify openness. The second reason is to assure that the box has the proper orientation of surface normals. The normals

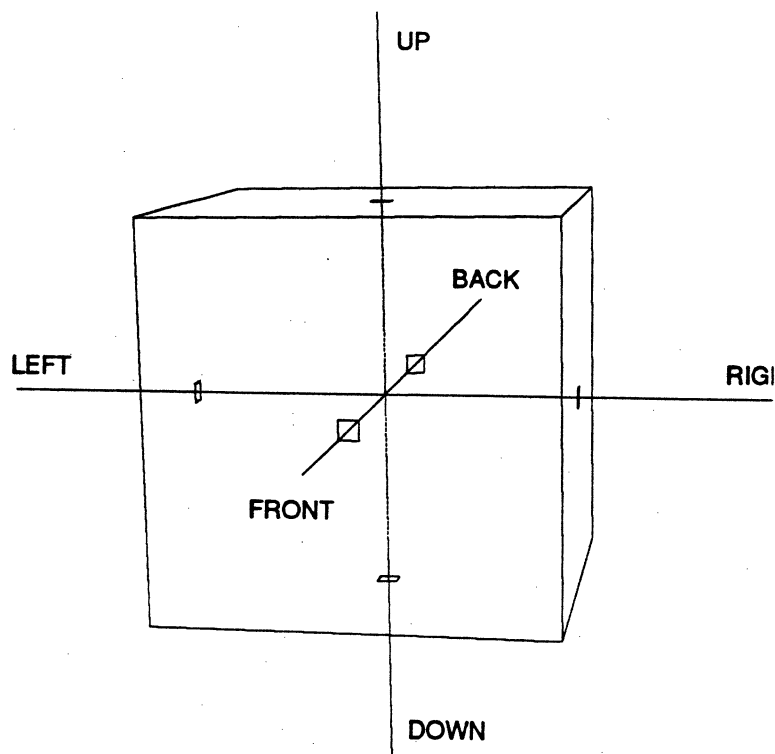


Figure 8-4: Naming Convention for Faces of a Box

must point inward in a right handed coordinate system in order for the box to be interpreted as a solid rather than a void in space. The convention adopted is shown in Figure 8-4.

To create the basic box (without rounded edges for now), use **rboxFrom6Planes**. When first thinking about rounded boxes, it is helpful to assign the plane values to variables which are named so that they reflect the geometry of the box (e.g., **XMinPlane**). Make the association between the geometry and the topological naming when invoking **rboxFrom6Planes** (e.g., put the **XMinPlane** in the "left" position).

**rboxFrom6Planes**( *Right, Left, Up, Down, Front, Back* )

**Returns** <rBox> Construct a rounded edged box from six planes. The returned box will not actually have rounded corners, the corners can be rounded with the **setRboxRadius** command.

*Right, Left, Up, Down, Front, Back*

<plane> The six faces of the box. These faces need not be perpendicular to one another.

The object returned by **rboxFrom6Planes** represents a box with 6 faces and sharp edges and corners. You can specify that one or more faces of the box are to be left open with **setRboxFacesOpen**. The face is indicated by one of the values '**RFACE**', '**LFACE**', '**UFACE**', '**DFACE**', '**FFACE**', '**BFACE**' which denote the right, left, up, down, front and back faces respectively. The faces argument may also be a list of those faces, or Nil to specify that none of the faces are to be open.

**setRboxFacesOpen**( *Rbox, Faces* )

**Returns** <Nil> Specify that certain faces of a rounded box are open.



**Rbox**        <rBox> The rounded box to modify.  
**Faces**       <keyword> The faces to be left open, must be one of 'RFACE, 'LFACE, 'UFACE, 'DFACE, 'FFACE, 'BFACE.

To round an edge, a radius is associated with the edge using **setRboxRadius**. The edge is indicated by a pair of the face letters in the order (R, L, U, D, F, B) followed by "edge". Thus the possible values for *Edge* are 'RUedge, 'RDedge, 'RFedge, 'RBedge, 'LUedge, 'LDedge, 'LFedge, 'LBedge, 'UFedge, 'UBedge, 'DFedge, 'DBedge. The 'RUedge is the one joining the "right" and the "up" faces. You may also specify the "edge" to be one of the face specifiers as given for **setRboxFacesOpen** above. In this case, all four edges of the specified face are set to the given radius. You may also use the keyword 'AllEdges, in which case each edge of the box is set to the specified radius. (Note: Unless you plan to change some of the radii, this is currently not useful because of a bug which does not allow all three radii meeting in a corner to be the same.)

**setRboxRadius( Rbox, Edge, Radius )**

**Returns**    <Nil> Set the radius for the edges of a rounded box.  
**Rbox**        <rBox> The rounded box to modify.  
**Edge**        <keyword> The edges to set. All the edges on a particular face can be specified with the face keyword ('RFACE, 'LFACE, 'UFACE, 'DFACE, 'FFACE, and 'BFACE). Individual edges adjacent to the right face are 'RUedge, 'RDedge, 'RFedge, and 'RBedge. Edges adjacent to the left face are 'LUedge, 'LDedge, 'LFedge, and 'LBedge. The remaining four edges are 'UFedge, 'UBedge, 'DFedge and 'DBedge. All the edges can be specified with 'AllEdges.  
**Radius**      <number> The radius of the selected edges.

To access the surfaces (contained in a shell object) which are used to represent the rounded box, you can use the geometry field of the object, just as for the other primitives:

**RboxObj->geometry**

Figure 8-5 shows the rounded box formed as a result of the following construction:

```
BasePlane := planeThruPtWithNormal( Origin, ZDir );
HeightPlane := reversePlane planeOffsetbyDelta( BasePlane, 0.5 );

LeftEndPlane := planeThruPtWithNormal( pt( -0.8, 0, 0 ), XDir );
RightEndPlane := reversePlane planeOffsetByDelta( LeftEndPlane, 1.6 );

BackPlane := planeThruPtWithNormal( pt( 0, 0.5, 0 ), vec( 0, -1, 0 ) );
FrontPlane := planeThruPtWithNormal( pt( 0, -0.5, 0 ), YDir );

% Form the basic box.
OuterCasing := rboxFrom6Planes( RightEndPlane, % Right
                                LeftEndPlane,   % Left
                                HeightPlane,     % Up
                                BasePlane,       % Down
                                FrontPlane,      % Front
                                BackPlane )$      % Back

% Set up the rounded edges. Only the four vertical edges
% are rounded.
setRboxRadius( OuterCasing, 'RFedge, 0.1 );
```

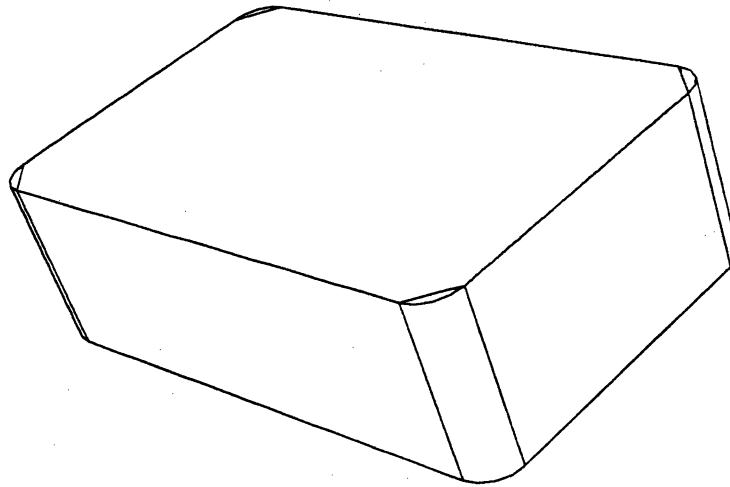


Figure 8-5: Rounded Edge Box

```
setRboxRadius( OuterCasing, 'REdge, 0.1 );
setRboxRadius( OuterCasing, 'LFedge, 0.1 );
setRboxRadius( OuterCasing, 'LBedge, 0.1 );
```

It is often useful to generate a number of related rounded-edge boxes. An operators for offsetting rboxes, **rboxOffsetFromRbox**, is used to generate a new rounded box from an existing one. If *Offset* is a scalar, the faces of the new rounded box are offset by that amount. Positive offsets go in the same direction as the plane normals, or towards the inside for a correctly defined box. The edge radii are adjusted so that a constant thickness is maintained. The radii are reduced for a smaller box, down to zero. Open faces remain open in the offset box.

More generally, if *Offset* is a list of "(facename . scalar)" pairs, each face is offset the respective amount. This can be used to produce more interesting results by making pairs of boxes which combine to produce solids with different wall thicknesses and opened sides. For example, consider a box with one face offset in a negative direction and the others offset in a positive direction. The "outer" box minus the "inner" box leaves a solid with an open face. This is different than constructing two boxes with an open face and then having to cap the edges surrounding the open faces.

**rboxOffsetFromRbox( Rbox, Offset )**

**Returns** <rBox> Construct a rounded box by offsetting from an existing rounded box.

**Rbox** <rBox> The reference rounded edge box.

**Offset** <number | listOf dottedPairOf keyword,number> The amount the new box is to be offset from the old. Positive values shrink the box towards its

center. If the list of dotted pairs is used the first value in the pair is one of the keywords 'RUedge, 'RDedge, 'RFedge, 'RBedge, 'LUedge, 'LDedge, 'LFedge, 'LBedge, 'UFedge, 'UBedge, 'DFedge, 'DBedge.

### 8.3 Interpolation & Approximation

Interpolation is used to create a curve or surface that possesses specified linear properties, usually position and derivative information. The most common use is to create a curve (or surface) passing through certain points, perhaps satisfying some derivative constraints. Other uses for interpolation arise in situations when theory predicts the existence of a curve or surface satisfying certain properties, but provides no way to compute it directly. For example, interpolation could be used for spline refinement and degree raising. Superior specialized algorithms have been developed to solve these particular problems, so interpolation is no longer used to solve them.

Several types of curve interpolation are so common that custom routines have been written to handle these cases. They are implemented as calls to a general curve interpolation package (described in the Programmer's Manual). If a new specialized curve application is desired, it can be implemented with the general curve interpolation. If it is used frequently, it should be encapsulated into a routine.

In general the routines take as arguments a list of parameters (or a parameter keyword), a vector of positions to be interpolated and possibly some additional information (e.g., end tangents). The positions argument specifies what data is to be interpolated. The parameters argument controls the parametrization of interpolant — at what parameter value the corresponding position value is interpolated. Different parametrizations can affect the shape of the resulting interpolant. Therefore, if you are dissatisfied with an interpolant, it might pay to experiment with the parametrization.

Possible parameter keywords include

Uniform  
ChordLength  
ParamByX  
ParamByY  
ParamByZ  
Radial

#### Complete Cubic Interpolation

The `completeCubicInterp` routine returns a cubic spline curve that goes through the positions (and optionally end tangents). End tangents are estimated if they are not provided. A Nil tangent specification will cause one to be estimated. Thus it is possible to supply one end tangent explicitly, and ask that the other be estimated.

**completeCubicInterp**( *Params*, *Positions*, *BeginTangent*, *EndTangent* )

**Returns**     <curve> Construct a curve interpolated from positions and two tangents.

**Params**     <listOf number | symbol> List of parameter values (in increasing order) or a parameter estimator keyword. For convenience the following symbols are provided, **Uniform**, **ChordLength**, **ParamByX**, **ParamByY**, **ParamByZ**, **Radial**.

**Positions**   <vectorOf point> Vector of positional data, one for each parameter.

**BeginTangent, EndTangent**

     <opt geomVector> These optional vectors indicate the direction of the curve at its end points. For convenience, either of these can be Nil.

We will use a single set of positional data (called "PosData") for interpolating in most of the examples throughout this section. Curves C1 and C2 are fit to that data, with estimated tangents and different parametrizations. For curve C3, end tangents are supplied explicitly. In curves C4 and C5, one tangent is specified, the other is to be estimated. Note that the magnitude of the tangent vectors can make an enormous difference. Having a direction in mind is not sufficient to get a good fit. In curve C6, the directions of the end vectors are the same as in the C3 example. However the magnitudes are 10 times as large, and the resulting curve is unsatisfactory. Curves C1 through C6 are shown in Figure 8-6.

```
PosData := vector( pt( -.8, .2 ), pt( -.5, -.4 ), pt( -.4, -.5 ),
                  pt( -.1, -.1 ), pt( .2, .3 ), pt( .5, .2 ),
                  pt( .7, 0 ), pt( .9, -.3 ) );

C1 := completeCubicInterp( Uniform, PosData );
C2 := completeCubicInterp( ChordLength, PosData );

C3 := completeCubicInterp( ChordLength, PosData,
                          vec( 2, 0 ), vec( 2, -2 ) );

C4 := completeCubicInterp( ChordLength, PosData, vec( 2, 0 ), Nil );
C5 := completeCubicInterp( ChordLength, PosData, Nil, vec( 2, 0 ) );

C6 := completeCubicInterp( ChordLength, PosData,
                          vec( 20, 0 ), vec( 20, -20 ) );
```

The `cubicCrvFromParametersAndPositions` function returns a cubic spline curve that goes through the positions. No tangent information is inferred or used. Knots are placed at all the parameters except the two nearest the ends. This is the not-a-knot condition for those familiar with interpolation. Curves C7 and C8 are two examples. The `crvFromParmInfoAndPositions` function returns a spline curve that goes through the positions. The interpolation occurs at the nodes for non-periodic splines. For periodic splines, the interpolation occurs at the knots. Curves C9 and C10 are examples for non-periodic and periodic splines respectively. Curves C7 through C10 are shown in Figure 8-7.

**cubicCrvFromParametersAndPositions( *Params*, *Positions* )**

**Returns** <curve> Construct a curve from a set of positions, no tangent information is used.

**Params** <listOf number | symbol> List of parameter values (in increasing order) or a parameter estimator keyword. For convenience the following symbols are provided, **Uniform**, **ChordLength**, **ParamByX**, **ParamByY**, **ParamByZ**, **Radial**.

**Positions** <vectorOf point> Vector of positional data, one for each parameter.

**crvFromParmInfoAndPositions( *ParmInfo*, *Positions* )**

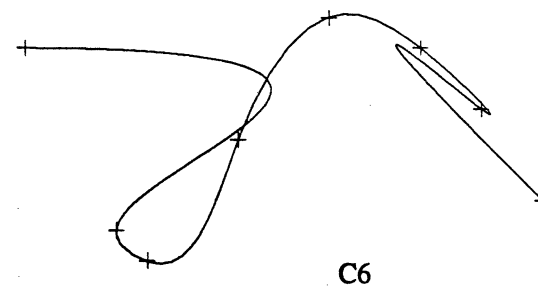
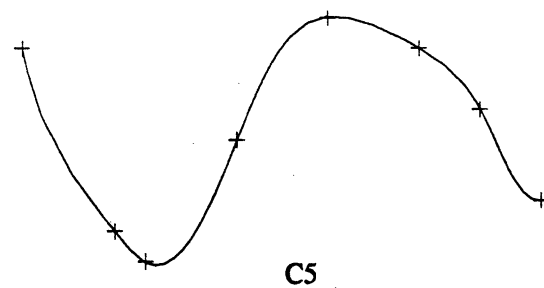
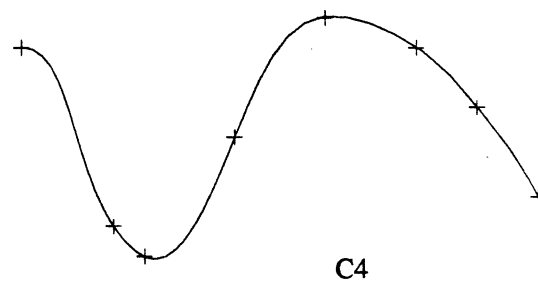
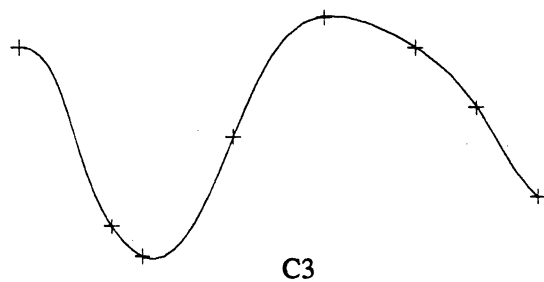
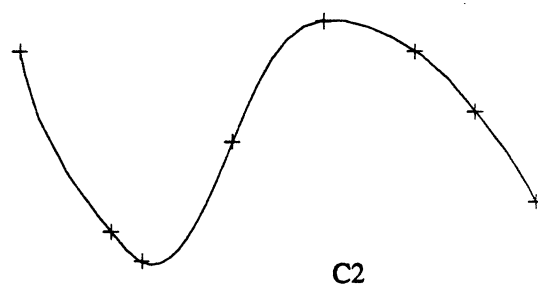
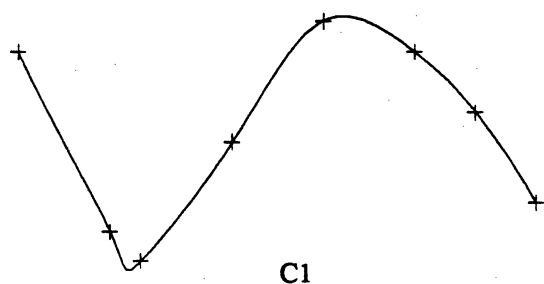
**Returns** <curve> Construct a curve given a parameter information structure and positions.

**ParmInfo** <parmInfo> A spline parmInfo.

**Positions** <vectorOf point> Vector of positional data, one for each parameter.

```
C7 := cubicCrvFromParametersAndPositions( Uniform, PosData );
```

```
C8 := cubicCrvFromParametersAndPositions( ChordLength, PosData );
```



**Figure 8-6: Complete Cubic Interpolation**

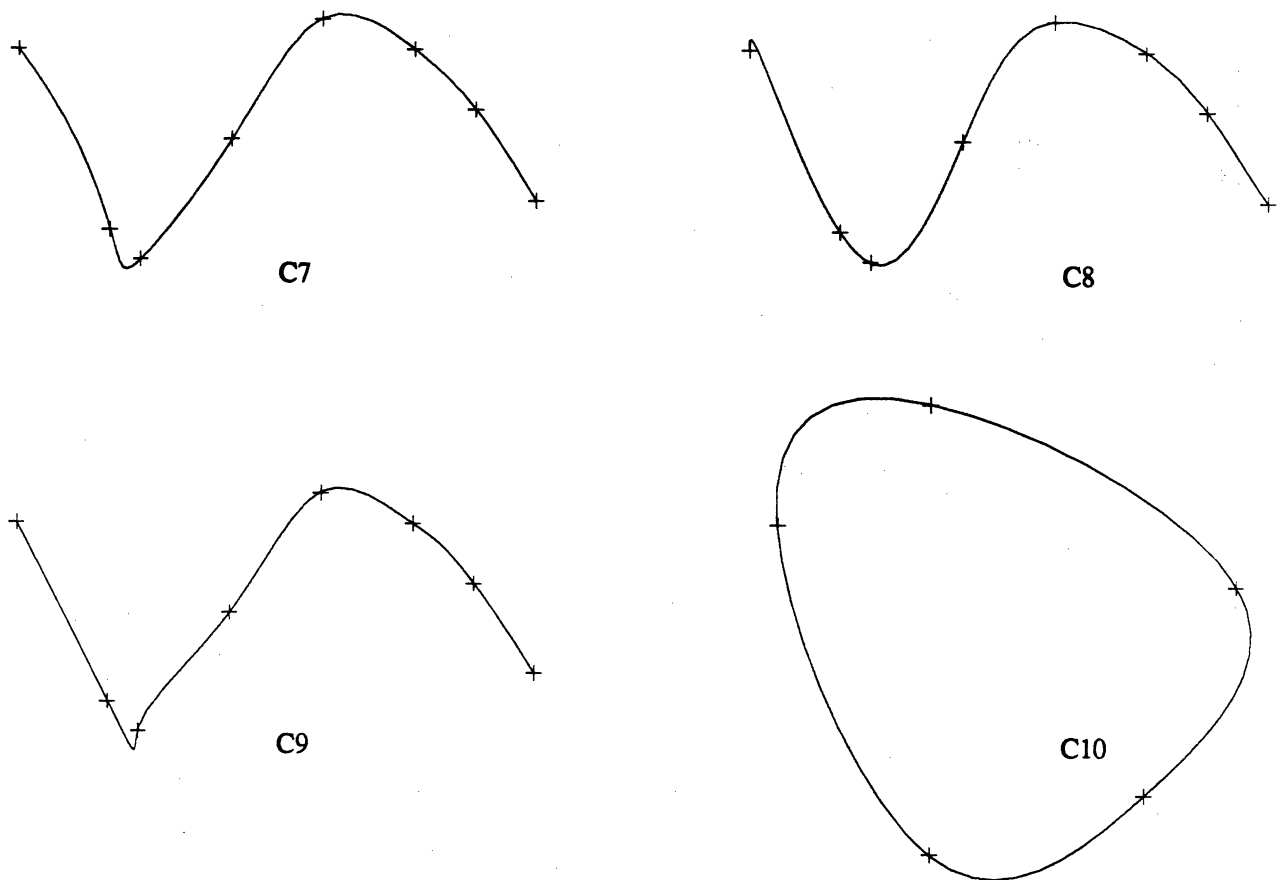


Figure 8-7: Various Curve Interpolations

```

C9 := crvFromParmInfoAndPositions(
    parmInfo( CUBIC, EC_OPEN, KV_UNIFORM ),
    PosData );

PeriodicData := vector( pt( 1, 0 ), pt( 0, .6 ),
    pt( -.5, .2 ), pt( 0, -.9 ),
    pt( .7, -.7 ));

C10 := crvFromParmInfoAndPositions(
    parmInfo( CUBIC, EC_PERIODIC, KV_UNIFORM ),
    PeriodicData );

```

Finally, **periodicCompleteCubicInterp** returns a cubic periodic spline curve that goes through positional data. If a list of parameters is given as the first argument (as opposed to a keyword), there should be one more parameter than the number of positions. This is to complete the circle. The principal use for *ExtraArg1* through *ExtraArgN* is to provide a center if the radial keyword is used to estimate the parameters. Curves C11 through C16 use various parametrizations provided by the fitting package. For radial parametrizations, it is possible to specify about which point to radially parametrize. This argument is optional. The default center is the origin, as used in curve C13. Curve C14 is the same curve, while a different center point for radial parametrization is used in

curve C15. Curve C16 uses parameters specified explicitly. Note that there are 6 parameters for 5 data points. Curves C11 through C16 are shown in Figure 8-8.

```
periodicCompleteCubicInterp( Params, Positions, ExtraArg1, ... )

Returns    <curve> Construct a periodic curve from parameters and positions.
Params    <listOf number | symbol> List of parameter values (in increasing order)
            or a parameter estimator keyword. For convenience the following symbols
            are provided, Uniform, ChordLength, ParamByX, ParamByY, ParamByZ,
            Radial.
Positions <vectorOf point> Vector of positional data, one for each parameter.
ExtraArg1, ...
            <point> Optional arguments for the parameter estimator.

C11 := periodicCompleteCubicInterp( Uniform, PeriodicData );
C12 := periodicCompleteCubicInterp( ChordLength, PeriodicData );

C13 := periodicCompleteCubicInterp( Radial, PeriodicData );

C14 := periodicCompleteCubicInterp( Radial, PeriodicData, Origin );

C15 := periodicCompleteCubicInterp( Radial, PeriodicData,
                                   pt(.2, -.2));

C16 := periodicCompleteCubicInterp( '(0 2 3 6 8 10)', PeriodicData );
```

## 8.4 Surface Construction Operations

This section describes a set of high-level operators for creating surfaces. Because most of the operators are very geometric in nature and do calculations using control points in the surfaces, some of them do not operate on rational splines. If you use primitives such as cylinders, or constructions involving arcs as beginning forms, you may need to refine them to a desired resolution before applying these operators. The operators which will not work will warn you if you attempt to use rational splines and will automatically convert to the non-rational form. The refinement is needed because the conversion changes the shape of the surface.

Most of the surface construction operators described in this section build surfaces from curve information. A few of them construct new surfaces which are derived from an existing surface (e.g., an offset surface).

### 8.4.1 Simple Surfaces

A very simple surface which is often useful as a starting point before applying other operations is just a flat surface in the XY-plane. The **flatSrf** operator constructs such a surface with specified orders and control mesh sizes so that the necessary degrees of freedom are available when needed (as opposed to just constructing a linear surface). The surface extends from -1 to 1 in both the X and Y directions. *Nrows* must be greater than or equal to *Vorder*, although this is not checked by the routine. A similar condition holds for the U direction (*Ncols* and *Uorder*). A related routine, **flatSrfBounds** allows specification of the X and Y extents of the resulting flat rectangular surface.

```
flatSrf( Uorder, Vorder, Nrows, Ncols )
```

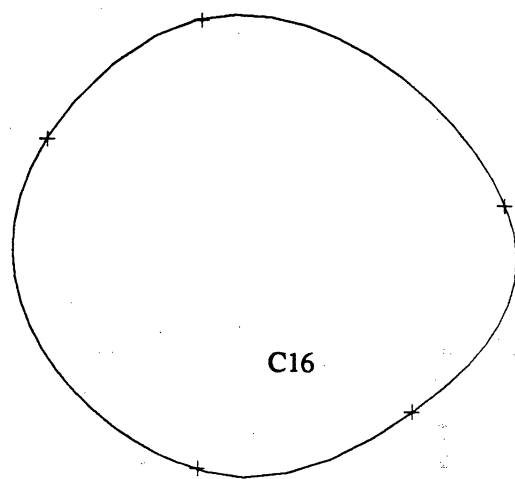
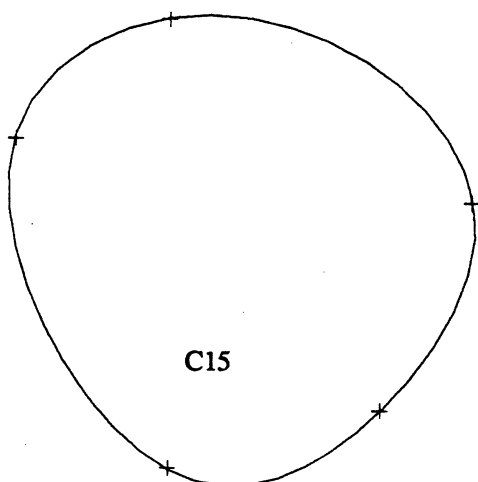
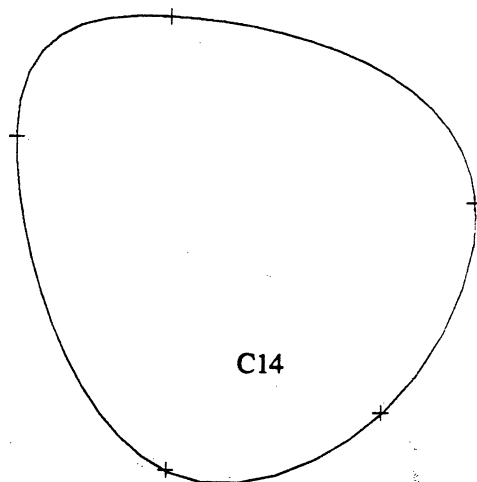
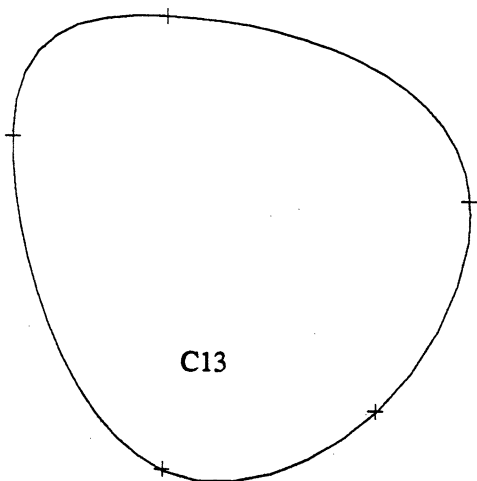
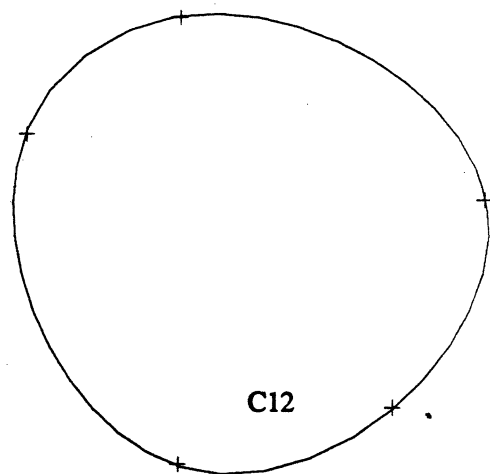
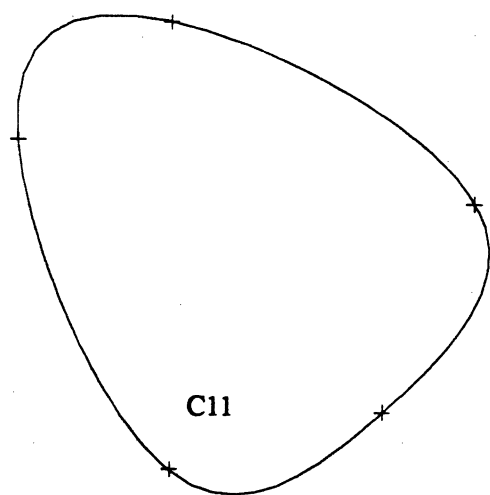


Figure 8-8: Periodic Curve Interpolation



**Returns** <surface> Construct a rectangular planar surface.  
**Uorder, Vorder** <number> Order of the surface in the U and V directions, respectively.  
**Nrows** <number> Number of control points in the V direction (equal to the number of rows in the control mesh).  
**Ncols** <number> Number of control points in the U direction (equal to the number of columns in the control mesh).

**flatSrfBounds( Uorder, Vorder, Nrows, Ncols, MinPt, MaxPt )**

**Returns** <surface> Construct a rectangular planar surface with the given bounds.  
**Uorder, Vorder** <number> Order of the surface in the U and V direction, respectively.  
**Nrows** <number> Number of control points in the V direction (equal to the number of rows in the control mesh).  
**Ncols** <number> Number of control points in the U direction (equal to the number of columns in the control mesh).  
**MinPt** <e2Pt> A point with X coordinate equal to the minimum X value of the resulting surface, and Y coordinate equal to the minimum Y value of the result.  
**MaxPt** <e2Pt> As for *MinPt*, except this point determines the other corner of the bounding box.

The **ruledSrf** operation constructs a ruled surface which has the two given curves as boundaries at opposite ends of the surface.

**ruledSrf( Crv1, Crv2 )**

**Returns** <surface> Construct a surface from two curves by connecting the curves with a series of straight lines.  
**Crv1, Crv2** <curve> The two boundary curves of the surface.

As an example of a ruled surface, the following sequence constructs the hyperboloid shown in Figure 8-9 (see section 9.2 [Transforming Objects], page 148 for an explanation of **ObjTransform**, an instancing operation):

```
Crv1 := objTransform( unitCircle, tZ( -1.0 ) )$
Crv2 := objTransform( Crv1, tZ( 2.0 ), rZ( 135 ) );
```

```
Srf := ruledSrf( Crv1, Crv2 )$
```

Coons' patches are a classical way to form surfaces from (curve) information at the boundaries of the surface. The **boolSum** operation is a special case of the more general Coons' patch. The boundary curves must be non-rational B-spline curves, and the blending is currently restricted to bilinear blending. A surface is produced which is the bilinearly blended Coons' patch defined by the four B-spline curves which are given as boundaries. The U pair of curves and the V pair of curves are made compatible if necessary so that each pair has the same orders and knot vectors.

**boolSum( Top, Bottom, Left, Right )**

**Returns** <surface> Construct a surface from four boundary curves.  
**Top** <curve> Curve on top boundary (V=0).  
**Bottom** <curve> Curve on bottom boundary (V=1).  
**Left** <curve> Curve on left boundary (U=0).

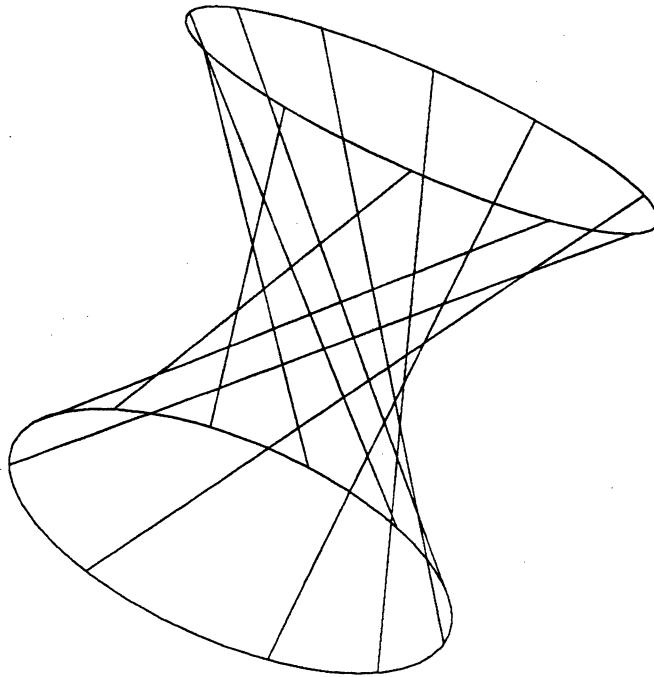


Figure 8-9: Ruled Surface

*Right*       $\langle \text{curve} \rangle$  Curve on right boundary ( $U=1$ ).

The **boolSum** operator assumes that the curves share endpoints as in Figure 8-10. The  $U$  parameter runs left to right;  $V$  runs top to bottom. The top and bottom curves should travel in the same direction, and the left and right curves should travel in the same direction. The top and left curves have the same first point, the bottom and right have the same last point. The last point of the top curve is the same as the first point of the right curve, and the first point of the bottom curve is the same as the last point of left curve.

Figure 8-11 shows four boundary curves and the resulting boolean sum surface.

### 8.4.2 General Sweeps

A *sweep* can be defined, in very general terms, as the generation of a curve, surface, or volume by moving some geometric object (a point, curve, surface, or volume) in 3-space. The union of all points in 3-space that are occupied by the object as it is moved becomes the resulting sweep entity. Sweeps in Alpha\_1 are a restriction of this general problem to the generation of surfaces by the sweeping of one 3-space curve (the *cross section curve*) along another 3-space curve (the *axis curve*). This operation is useful in that it reduces the problem of surface design to the simpler problem of designing one or more curves.

Sweeps in Alpha\_1 are further constrained by restricting the manner in which the cross section curve is oriented relative to the axis curve as it is swept along the axis curve. If the cross section curve lies in the  $Z = 0$  plane, it will be kept perpendicular to the axis curve as it is swept along it. In general, the  $Z = 0$  plane of the defining coordinate system for the cross section is kept perpendicular to the axis curve at all times as the cross section curve is swept.

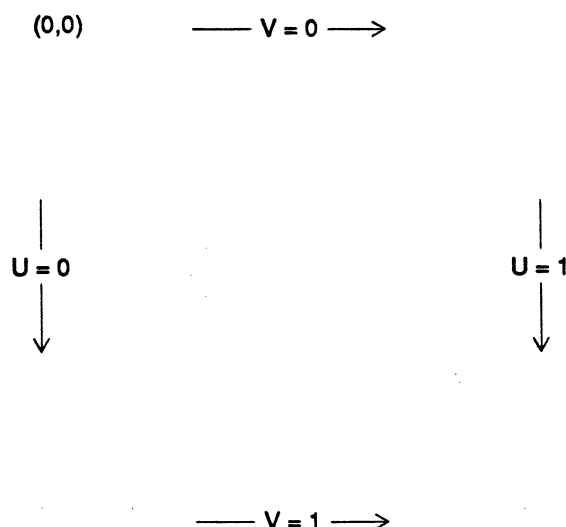


Figure 8-10: Orientation of Boundary Curves for Boolean Sum Surface

An *unblended sweep* involves the sweeping of a constant cross section along the axis curve. It is possible to change the scale of the cross section curve as it is swept (without changing its shape). This is accomplished with a *profile curve*. The profile curve is used to express the scaling of the cross section as a function of arc-length along the axis curve.

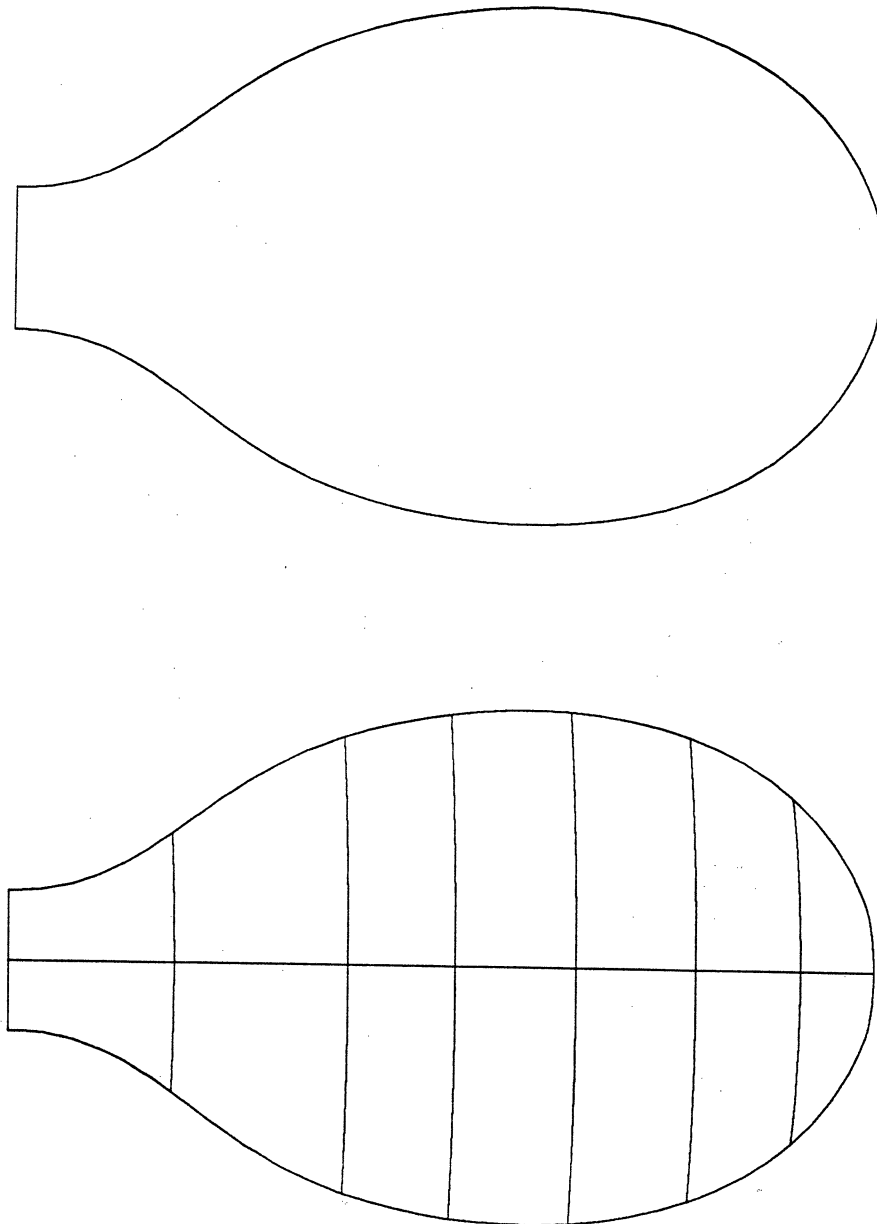
We can also allow the shape of the cross section to change in a more general way as it is swept along the axis curve. To do this, several cross section curves are specified along with blending information. The changing cross section, at any given point, will be a blend of the cross sections specified, so these are called *blended sweeps*.

#### Properties of the Simple Sweep Routines

Both the axis and cross section curves may be any arbitrary B-spline curves. The resulting sweep surface will inherit the parametric information from the axis curve in the  $U$  direction and the parametric information from the cross section curve in the  $V$  direction.

For unblended sweeps not involving profile curves, the accuracy of the resulting sweep surface does not depend at all on the nature of the cross section curve. However, the accuracy of the resulting surface depends a great deal on the nature of the axis curve.

A sweep of any cross section along an axis curve composed entirely of straight lines and arcs (that are unit tangent continuous) will yield an exact result. All other cases yield an approximate result, but the approximation may be made arbitrarily good with refinement of the axis curve. Sections of the axis curve where the curvature is changing rapidly require the most refinement to make the sweep surface converge to the correct result. Currently this refinement must be done by hand (eventually the sweep package may automatically refine the axis.)



**Figure 8-11:** Boundary Curves and Boolean Sum Surface

## 8.5 Unblended Sweeps

The simplest sweep, **circTubeConstantWidth**, requires only an axis curve and a radius to be specified.

**circTubeConstantWidth**( *Axis*, *Radius* )

**Returns** <surface> Creates a simple sweep of constant width and circular cross section.

**Axis** <curve> The curve along which the circular cross section is to be swept.

**Radius** <number> The radius of the circular cross section to use.

A slightly more general version, **circTubeWithProfile**, uses a profile curve to specify variable scaling of the circular cross section curve as it is swept along the axis curve.

**circTubeWithProfile**( *Axis*, *Profile* )

**Returns** <surface> Creates a sweep of variable width and circular cross section.

**Axis** <curve> The curve along which the cross section is to be swept.

**Profile** <curve> A curve specifying the scaling of the cross section as it is swept along the axis curve.

Profile curves should lie in the  $Z = 0$  plane and **must** be monotonic in  $X$  (i.e., profile curves must describe explicit functions of  $X$ ). No test is made to insure that this is the case. A mapping is established between the arc-length along the axis curve and the range of  $X$  values on the profile curve. At any given point along the axis curve, the value of  $Y$  at the associated value of  $X$  on the profile curve is used directly as the factor by which the cross section is scaled.

The value of the **QRefMeasure!**\* variable determines the amount of refinement used on the profile curve. Refinement of the profile curve is needed to insure that the geometry of the profile curve is accurately reflected in the resulting sweep. If the geometry of the profile curve is not properly reflected in the resulting sweep surface, then the value of the **QRefMeasure!**\* variable should be lowered. Currently the values of this variable are not very intuitive to change. If you wish to do the refinement on the profile curve yourself, set the **QRefMeasure!**\* variable to Nil. As a general rule, the profile curve should be refined in areas of high curvature.

**QRefMeasure!**\*

<number> Determines the amount of refinement used for profile curves (default 50).

Two other sweep functions, **sweepConstantWidth** and **sweepWithProfile**, also use only a single cross section curve, but allow that cross section to be specified by the user. Cross section curves may be any arbitrary B-spline curves. As a general guideline, the most intuitive sweeps result from cross sections that are defined in the  $Z = 0$  plane and are "centered" about the origin. At times it may be desirable to use non-planar cross section curves. One should be aware, however, that results may sometimes be a bit unexpected. The resulting sweep surface may not be unit tangent continuous even though the axis and cross section curves are.

**sweepConstantWidth**( *Axis*, *CrossSec*, *Scale*, *XInNormPlane* )

**Returns** <surface> Creates a sweep of constant width and arbitrary cross section.

**Axis** <curve> The curve along which the cross section is to be swept.

**CrossSec** <curve> The cross section curve that is to be swept along the axis curve.

**Scale** <number> The cross section is to be scaled by this constant value before it is swept along the axis curve.

**XInNormPlane**

<geomVector | Nil> A vector giving the orientation to be used for the cross section at the first point of the axis curve.

**sweepWithProfile**( *Axis*, *CrossSec*, *Profile*, *XInNormPlane* )

**Returns** <surface> Creates a sweep of variable width and arbitrary cross section.  
**Axis** <curve> The curve along which the cross section is to be swept.  
**CrossSec** <curve> The cross section curve that is to be swept along the axis curve.  
**Profile** <curve> A curve specifying the scaling of the cross section as it is swept along the axis curve.

**XInNormPlane**

<geomvec or Nil> A vector giving the orientation to be used for the cross section at the first point of the axis curve.

The *XInNormPlane* argument is used to completely specify the orientation of the cross section curve at the first point of the axis curve. The first cross section is positioned so that the origin of the coordinate system in which the cross section curve is defined is translated to the first point of the axis curve and the Z axis of the coordinate system in which the cross section curve is defined is aligned with the unit tangent at the first point of the axis curve. (This results in the first cross section being positioned in the normal plane at the first point of the axis curve if the cross section is defined in the  $Z = 0$  plane.)

If an *XInNormPlane* vector argument is specified, then this first cross section is oriented with the X axis of its defining coordinate system aligned with the *XInNormPlane* vector projected onto the normal plane at the first point of the axis curve.

If this argument is Nil, then the first cross section is oriented with the X axis of its defining coordinate system aligned with the principal normal to the axis curve at this point. If the principal normal is not defined for the first point of the axis curve (e.g., the first interval of the curve is linear), then the first point on the curve that does have a principal normal is located and this normal is "mapped" back to the beginning of the curve and used to orient the first cross section.

If all else fails (e.g., the curve is composed entirely of straight line segments) then an arbitrary vector in the normal plane of the first point of the curve is used in place of a normal to orient the first cross section.

### Things to Keep in Mind

A list of useful things to keep in mind when using the sweep package includes:

- Both the cross section and axis curves may be any arbitrary B-spline curves.
- The resulting sweep surface will inherit the parametric information from the axis curve in the U direction and the parametric information from the cross section curve in the V direction.
- It is necessary to refine the axis curve by hand in areas of rapidly changing curvature. It is not necessary to refine the axis curve on sections that are straight lines or arcs (where the curvature is constant).
- If a profile curve is used, it will be automatically refined in areas of high curvature so that its geometry will properly be reflected in the resulting sweep surface. The amount of refinement done on the profile curve is determined by the **QRefMeasure!**\* variable described above. If you wish to do the refinement of the profile curve by hand, set the **QRefMeasure!**\* variable to Nil, then use **cRefine** to do the refinement of the profile curve in areas of high curvature.
- If cross section blending is specified (to be described in the next section), refinement of the axis curve according to the cross section placement is necessary. This refinement is done automatically by default. If you wish to do the refinement by hand, set the **QCrossSpread!**\*

variable to Nil, then use **cRefine** to do the refinement of the axis curve about the parametric values corresponding to each cross section placement.

## 8.6 Blended Sweeps

The most general sweep operator, **generalSweep**, allows the use of a profile curve to control the scaling of the cross sections as well as a number of different cross sections with several blending options. The cross section used at any given point will be a blend of the cross sections specified, scaled in accordance with the profile curve (if one is given).

**generalSweep( Axis, CrossSec, BlendFlag, Profile, XInNormPlane )**

<b>Returns</b>	<surface> Creates a sweep using blending of cross sections and/or variable scaling.
<b>Axis</b>	<curve> The curve along which the cross section is to be swept.
<b>CrossSec</b>	<curve   listOf listOf curve,number> A curve specifying the cross section desired or a list containing cross section curves and blending information. Each element of the list is a list whose first element is a cross section curve and whose second element is a floating point value. The floating point value indicates the placement of that cross section curve along the axis curve in a manner controlled by the <i>BlendFlag</i> argument. Currently, the only type of blending used is linear interpolation between cross sections.
<b>BlendFlag</b>	<keyword   Nil> If the <i>CrossSec</i> argument is a curve, then this argument should be Nil. Otherwise it should be one of 'ARC_LENGTH_BLEND or 'PARAM_BLEND.
<b>Profile</b>	<curve   Nil> A curve specifying the scaling of the cross section as it is swept along the axis curve. No variable scaling is specified if this argument is Nil.
<b>XInNormPlane</b>	<geomVector   Nil> A vector giving the orientation to be used for the cross section at the first point of the axis curve.

If the *BlendFlag* is 'ARC\_LENGTH\_BLEND then the floating point values in the *CrossSec* represent normalized arc-length measures on the axis curve where the corresponding cross section should be interpolated. If the *BlendFlag* is 'PARAM\_BLEND, the floating point values represent parametric values on the axis curve where the corresponding cross section should be interpolated.

When specifying the blending of cross sections, it should be noted that the cross sections specified in the *CrossSec* argument to **generalSweep** will not in general be interpolated by the resulting sweep surface. Rather, these cross sections will be approximated at the points specified along the axis curve.

It is usually necessary to refine the axis curve in the vicinity of each cross section placement in order that the resulting sweep take on the characteristics of the cross sections specified. By default this refinement is done automatically. The scheme that is used to do the automatic refinement to cross section placement is very simple and will probably not always do what you want. In fact, it is not possible to know exactly what shape is intended by the cross section placements. For this reason you may want to do the refinement to cross section placement by hand.

The **QCrossSpread!\*** global variable controls automatic refinement to cross section placement. If **QCrossSpread!\*** is set to something other than Nil, then the sweep routine will automatically do refinement to cross section placement. This variable helps control how closely the resulting sweep surface will come to interpolating the cross sections specified. The **QCrossSpread!\*** variable may

take any value between 0.0 and 1.0. If the value of **QCrossSpread!\*** is zero, then the specified cross sections will be interpolated; however the resulting surface will only be C-zero at the points of interpolation. (A C-zero surface does not have continuous tangent vectors, so it may contain "sharp" edges.) To avoid this C-zero behavior, raise the value of **QCrossSpread!\***. By default the **QCrossSpread!\*** variable is set to a non-Nil value (0.1).

#### **QCrossSpread!\***

**<float>** Determines the amount of refinement of the axis curve for cross section placement (default 0.1).

To specify refinement to cross section placement by hand, set **QCrossSpread!\*** to Nil, then use the **cRefine** routine to refine the axis curve in the parametric vicinity of each cross section placement.

Unfortunately, it is also possible to over-refine the axis curve when working with multiple cross sections. This is because the only type of cross section blending currently done is linear interpolation between cross sections. If the axis curve is over-refined, then this linear interpolation will become apparent in the resulting sweep surface (i.e., the surface will begin to look piecewise linear in the axis curve direction).

For those who want more details, what **QCrossSpread!\*** really controls is a refinement of the axis curve to the cross section placement. Such a refinement is in general necessary if the resulting sweep surface is to take on the characteristics of the specified cross sections.

The idea is as follows. If a knot of multiplicity Order-1 is inserted at each point indicated for the cross section placements, then each cross section will be interpolated. Unfortunately, the surface will be only C-zero at such points. Instead, the knot of multiplicity Order-1 can be "spread out" over some portion of the parametric domain of the knot vector (by adding Order-1 uniformly spaced knots over that portion of the parametric domain). The sweep surface will then come close to each cross section indicated, without interpolation. Some smoothing properties will thereby be retained. The **QCrossSpread!\*** variable specifies the fraction of the parametric domain of the axis curves' knot vector over which each "multiple" knot is to be spread.

## 8.7 Sweep Examples

This section describes a set of examples constructed with the sweeping functions.

We can easily build a torus by using the **circTubeConstantWidth** routine, specifying a circle to be swept around the unit circle. (Of course, this is not really necessary, since **shape\_edit** contains a torus primitive.)

```
Torus ^= circTubeConstantWidth( UnitCircle, .1 )$
```

The next example illustrates sweeping a circle along an axis curve composed entirely of straight lines and arcs. Note that the resulting sweep surface is exact (Figure 8-12).

```
LLine := lineVertical( -.5 )$
RLine := lineVertical( .5 )$
MidLine := lineHorizontal( 0 )$
LArc := arcRadTan2Lines( .25, LLine, MidLine )$
RArc := arcRadTan2Lines( .25, MidLine, RLine )$
RotRArc := objTransform( RArc, rx 90 )$

ArcsAndLin ^= profile( LArc, RotRArc )$
```



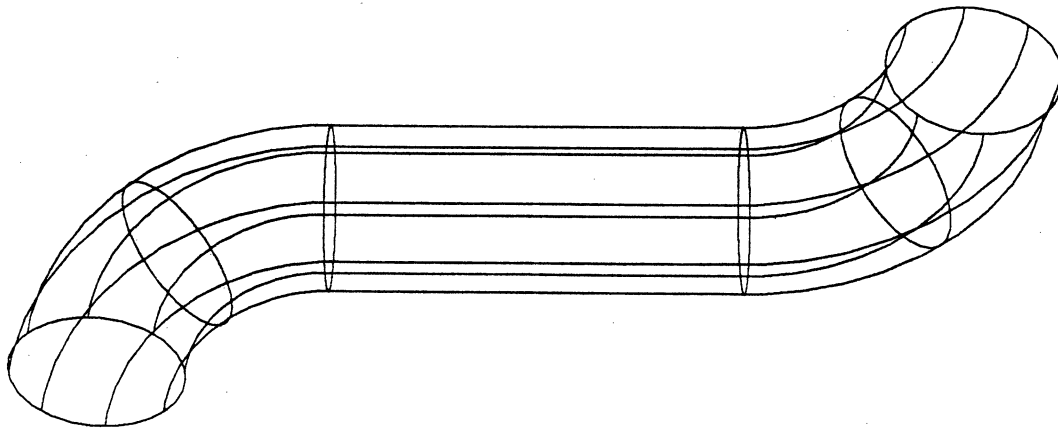


Figure 8-12: Circular Cross Section Sweep

```
ArcsAndLinSrf1 ^= circTubeConstantWidth( ArcsAndLin, .1 )$
```

The same axis curve can be used with a more general cross section. In Figure 8-13 a cross section curve (GenCrv) is shown, and Figure 8-14 shows the resulting sweep surface. Again we note that the resulting sweep surface is exact.

```
GenCrvPt0 := pt( -.1, -.7 )$
GenCrvPt1 := pt( -.6, .2 )$
GenCrvPt2 := pt( .1, -.2 )$
GenCrvPt3 := pt( .2, .2 )$
GenCrvPt4 := pt( .6, .5 )$
GenCrvPt5 := pt( .8, .0 )$
```

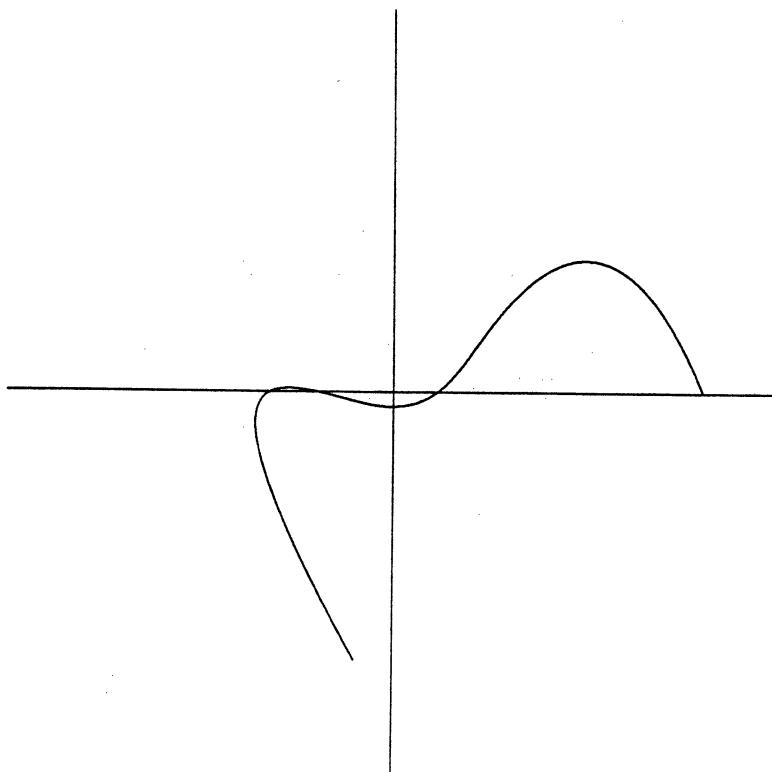
```
GenCrv ^= curve( parmInfo( 4, EC_OPEN, KV_UNIFORM ),
  list( GenCrvPt0, GenCrvPt1, GenCrvPt2,
    GenCrvPt3, GenCrvPt4, GenCrvPt5 ) )$
```

```
ArcsAndLinSrf2 ^= sweepConstantWidth( ArcsAndLin, GenCrv, .2, NIL )$
```

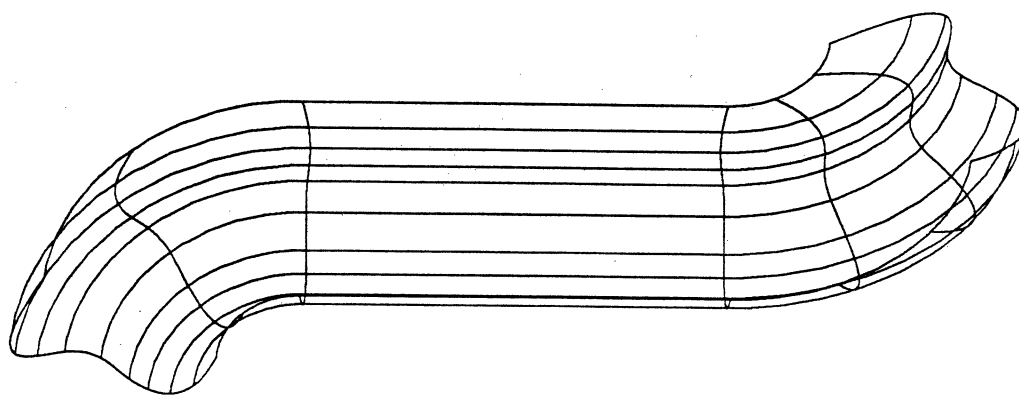
We now make a sweep using a more general axis curve. The curve that was previously used as a cross section curve will now be used as an axis curve. The resulting sweep is **not** exact (Figure 8-15). But if we refine the curve everywhere, we get a much better approximation (Figure 8-16). Note that the surface really is self-intersecting in this case.

```
GenCrvSrf ^= circTubeConstantWidth( GenCrv, .1 )$
```

```
% First look at the knot vector so we know how to refine it.
```



**Figure 8-13: Cross Section Curve**



**Figure 8-14: Constant Width Sweep**

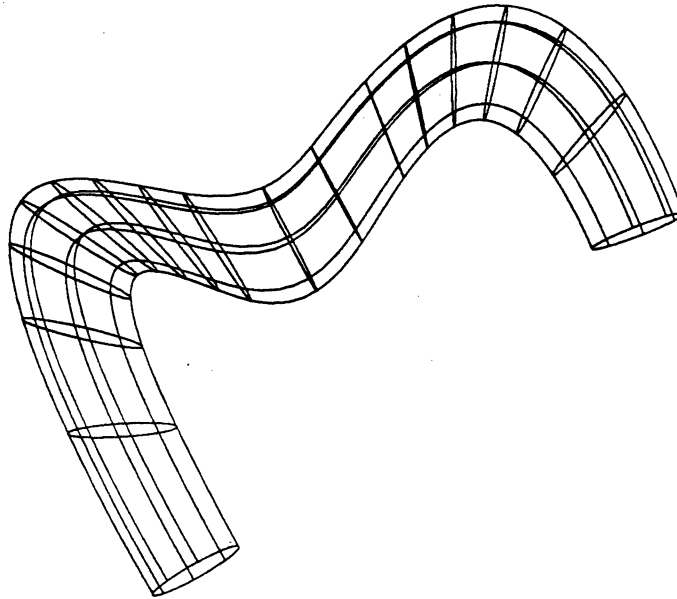


Figure 8-15: General Axis Curve Sweep

```

cKv GenCrv;

% Now refine it.
GenCrvRKnotVec := append( append( list( 0.0, 0.0, 0.0 ),
    numRamp( 31, 0.0, 3.0 ) ),
    list( 3.0, 3.0, 3.0 ) )$
RGenCrv := cRefine( GenCrv, GenCrvRKnotVec, NIL )$

% The resulting sweep is much more accurate.
RGenCrvSrf ^= circTubeConstantWidth( RGenCrv, .1 )$

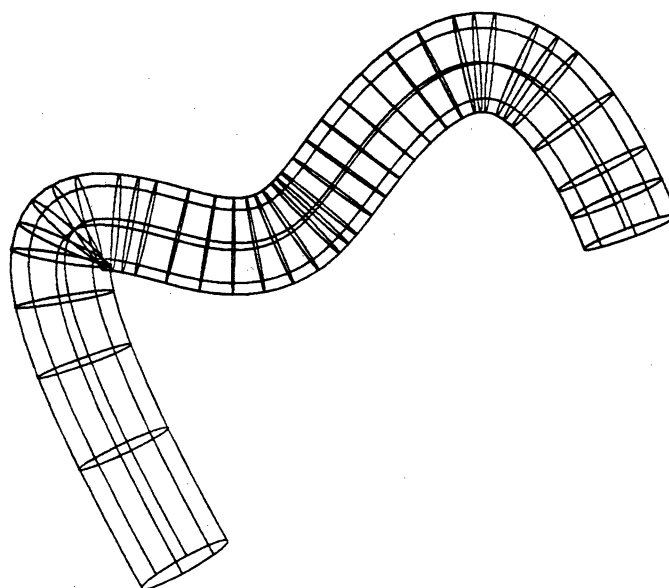
```

The use of a profile curve is demonstrated by making a simple goblet as a variable scaling of a square cross section that is swept along a linear axis curve. The profile curve is shown in Figure 8-17, and the resulting sweep surface is shown in Figure 8-18. The square goblet might be a bit awkward to use (the wine will tend to dribble out the corners), so the next example uses a more sophisticated sweep to get a more satisfactory goblet. The result is shown in Figure 8-19.

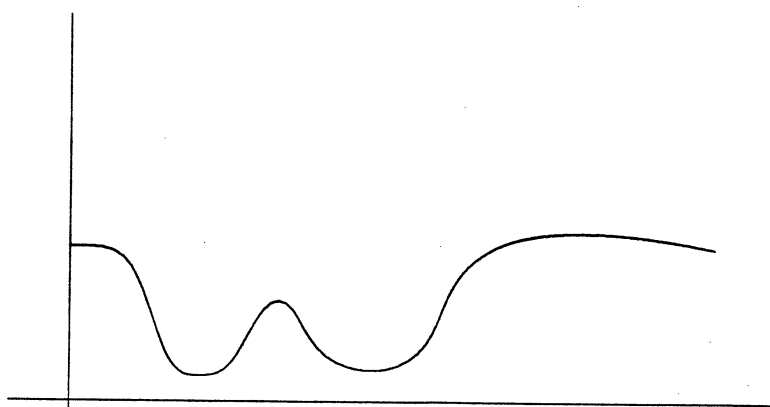
```

% Create a goblet shaped profile curve.
GobPt1 := pt( 0.0, 0.24 );
GobPt2 := pt( 0.05, 0.24 );
GobPt3 := pt( 0.1, 0.24 );
GobPt4 := pt( 0.15, 0.04 );
GobPt5 := pt( 0.2, 0.04 );
GobPt6 := pt( 0.25, 0.04 );
GobPt7 := pt( 0.30, 0.16 );
GobPt8 := pt( 0.35, 0.16 );

```



**Figure 8-16:** General Axis Curve Sweep With Refinement



**Figure 8-17:** Goblet Profile Curve

```

GobPt9  := pt( 0.39, 0.04 );
GobPt10 := pt( 0.6,  0.04 );
GobPt11 := pt( 0.6,  0.32 );
GobPt12 := pt( 1.0,  0.24 );

% Note that the profile curve represents an explicit function of X.
GobletProfile ^= curve( parmInfo( CUBIC, EC_OPEN, KV_UNIFORM ),
    list( GobPt1, GobPt2, GobPt3, GobPt4, GobPt5, GobPt6,
        GobPt7, GobPt8, GobPt9, GobPt10, GobPt11, GobPt12 ) );

% We'll give the goblet a square cross section.
SqrParm := parmInfo( linear, EC_OPEN, KV_UNIFORM );
UnitSqrSec := curve( SqrParm,
    list( pt( 1, 1 ), pt( -1, 1 ),
        pt( -1, -1 ), pt( 1, -1 ), pt( 1, 1 ) ));

% Use a simple linear axis.
% We degree raise the axis to cubic since we don't want the resulting
% sweep surface to be piecewise linear in the u (axis) direction.
GobletAxis := raiseOrder( profile( pt(0,0), pt(1,0) ), cubic )$

SqrGoblet ^=
    sweepWithProfile( GobletAxis, UnitSqrSec, GobletProfile, NIL )$

% We'll blend from a square to a circular cross section.
% We need to rotate the square cross section so as to align it with
% the circular cross section (otherwise we'll get a nasty twist in
% the resulting sweep).
RotUnitSqrSec := objTransform( UnitSqrSec, rz( -45 ) );

% The specification below indicates that as we sweep along
% the axis curve: from arc length value 0.0 to 0.1 the square cross
% section should be used. From 0.1 to 0.15 a blend of the square and
% circle cross sections should be used. From 0.15 to the end of
% the curve the circular cross section should be used.
SectionCrvs := list( list( RotUnitSqrSec, 0.1 ),
    list( UnitCircle, 0.15 ) )$

Goblet ^=
    generalSweep( GobletAxis, SectionCrvs, 'ARC_LENGTH_BLEND,
        GobletProfile, NIL )$

```

## 8.8 Derived Surfaces

The `vOffset` operator provides an approximation to the offset surface of a given surface. Exact offsets to B-spline surfaces cannot in general be represented as B-spline surfaces, so some approximation is required. The accuracy of this approximation depends on the original surface. A highly-curved surface should probably be refined somewhat in order to produce a reasonable approximation to the offset surface. There are no guarantees about the behavior of the resulting surface. In particular, self-intersecting surfaces are quite simple to generate. It is also wise to pay attention

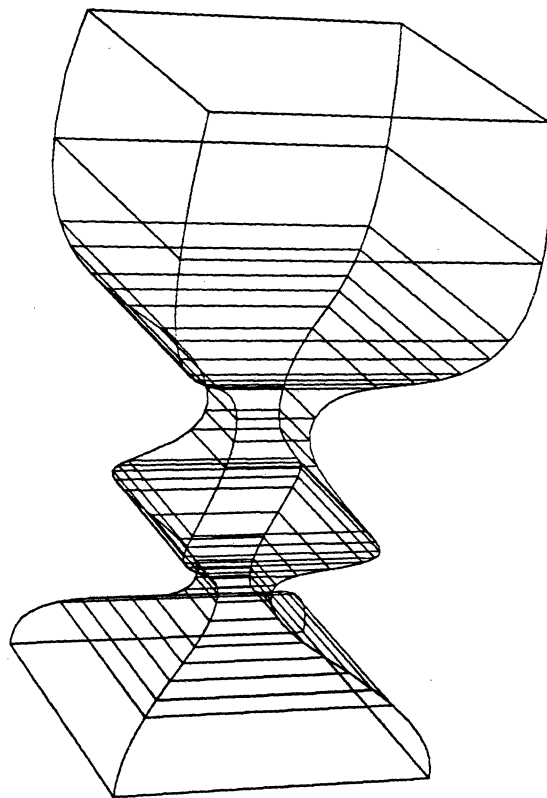
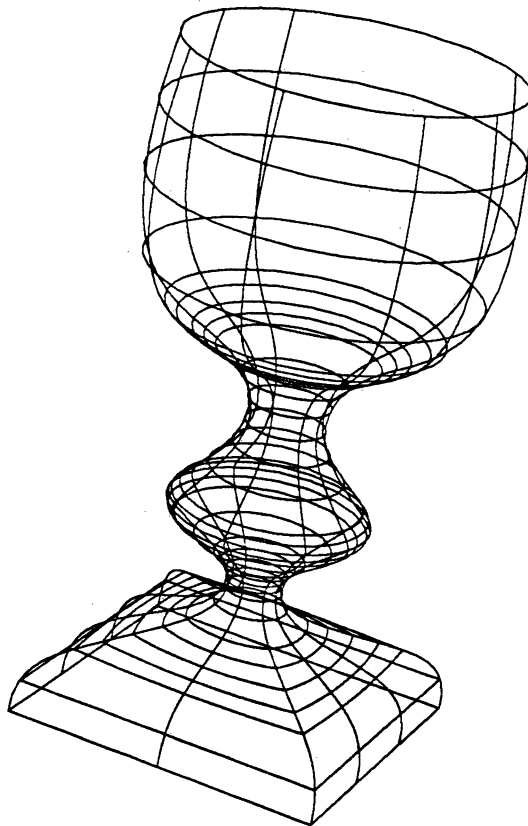


Figure 8-18: Square Cross Section Goblet





**Figure 8-19:** Blended Cross Section Goblet



to the orientation of the surface, as the sign of the offset distance must change if the surface is not oriented in the expected direction. The operator does not necessarily handle embedded discontinuities (e.g., *cusps*) gracefully.

The offset direction is taken to be the normal to the surface at the node values corresponding to each of the control points. This may not always be a good choice, although it should be reasonable for "well-behaved" surface parametrizations. The simplest offsets are those which offset each point by a constant distance. More complex, but often useful, are variable offsets. These result in surfaces in which the control points are offset from the original by varying distances.

**vOffset( Srf, DistInfo )**

<b>Returns</b>	<surface> Construct a surface offset from another surface.
<b>Srf</b>	<surface   listOf surface> Surface to be offset (or list of surfaces).
<b>DistInfo</b>	<number   table   function> Information describing the offset distance. It may be one of three types:
Constant	Each control point is offset by a constant distance.
Table	Each control point is offset by a distance given in the table. The table is assumed to have the same dimensions as the control mesh.
Function	Each control point is offset by a distance determined by calling the given function with the indices of the point and the geometric coordinates.

A variation of the **vOffset** operator is to restrict the direction of offset of each control point to be a constant, rather than an associated surface normal. This results in a lifting operation (or projection, depending on your point of view). The **lift** operator produces an approximation to the lifting of the given surface based on the distance specifications. The surface to be lifted will generally be flat, although it doesn't have to be, and the distance will typically be given as a function of the control points, although constant lifting and table lookup are also allowed. Using a constant is the same as translating the surface along the specified direction vector.

**lift( Srf, Direction, DistInfo )**

<b>Returns</b>	<surface> Construct a surface by lifting another surface.
<b>Srf</b>	<surface> Surface to be lifted.
<b>Direction</b>	<geomVector> Direction vector of the lift; each control point in the original surface is offset in this direction.
<b>DistInfo</b>	<number   table   function> Specification for how far to offset each control point (as in <b>vOffset</b> above).

As an example, Figure 8-20 shows the results of applying a **lift** and then an **vOffset** operation to a flat surface.

```
% Define a procedure which will return the distance of
% the i,j-th control point from the origin.
procedure parabFn( S, Index1, Index2 );
    distPtPt( S->sMesh[Index1][Index2], Origin );

% Offset direction will be along the Z axis.
Dir := ZDir;

% Start with a flat surface.
```

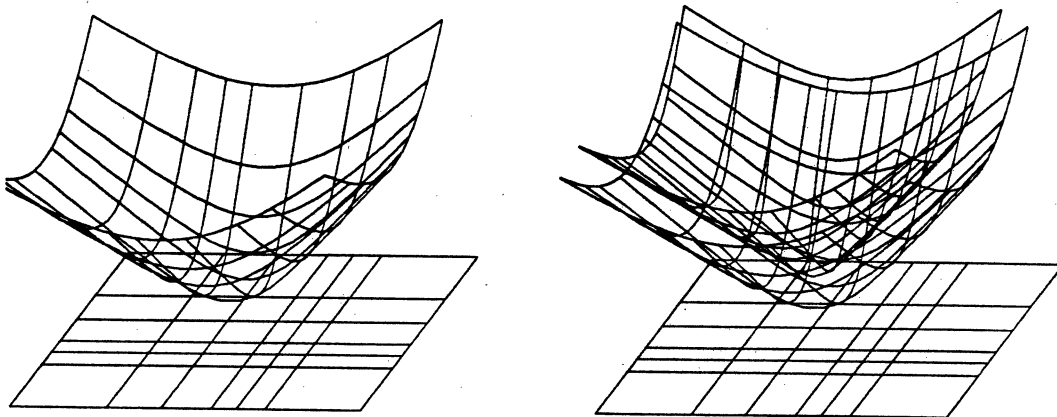


Figure 8-20: Example of lift and vOffset Operations

```
TestSrf := flatSrf( 4, 4, 8, 8 )$

% Make a parabola-like surface using the lift operator.
ParabolaSrf1 := lift( TestSrf, Dir, 'parabFn )$

% Offset the first surface.
ParabolaSrf2 := vOffset( ParabolaSrf1, -0.2 )$
```

It is often useful to construct a surface which joins or closes a gap between two existing surfaces. The **vOffset** operator, for example, produces from one surface an offset surface, but does not produce enough surfaces to form the boundary of a solid object. The **srfEdge** operator addresses this issue. The resulting surface will form a band which connects all four sides of one surface to the other.

**srfEdge( Srf1, Srf2, EdgeType, TangInfo )**

**Returns** <surface> Construct a surface joining two other surfaces by connecting their four boundary curves.

**Srf1, Srf2** <surface> The two surfaces to be joined.

**EdgeType** <keyword> Either **'LINEAR** or **'CUBIC**, signifying the order of the cross-section of the edge.

**TangInfo** <listOf number> If **EdgeType** is **'CUBIC**, then two tangent weights must be given: one for how much tangency going from the surface to the edge, and one for how much tangency going around the corners. For **'LINEAR** **EdgeType**, this argument is ignored.

The pair of surfaces being joined is assumed to be "related" so that they have the same knot

vectors and the  $i,j$ -th control point of one corresponds to the  $i,j$ -th control point of the other. These conditions commonly hold if one of the surfaces is an offset or translation of the other. Currently the edge constructed is either linear or cubic in one parametric direction. The other direction (which wraps around the surfaces) depends on the order of the surface. Note: The cubic edge type currently only works for bi-cubic surfaces due to the way in which the operator handles the problem of getting around the corners.

In the linear edge construction, the boundary points of the two surfaces are used directly to construct the edge surface. If the cubic edge construction is chosen, the boundary points are also used. In addition, a tangent vector is computed at each point on the surface boundary (the tangent going across the boundary), and this tangent is used to construct another point interior to the edge surface in such a way that the edge surface comes in tangent to the original surface. The first tangent weight given in the *TangInfo* list is just a scale factor which is applied to the cross-boundary surface tangent before offsetting from the boundary point. A value between 0.5 and 1.0 is a good choice to begin with; larger values will tend to stretch out the joining edge band and smaller values will tend to shrink it to a "sharper" (though still continuous) edge. The corners are handled in exactly the same way, with the tangency being determined from the parts of the edge surface which have already been computed. The second tangent weight from the *TangInfo* list should probably not exceed 1.0 by a great amount.

As a variant of the *srfEdge* operator, the *singleSrfEdge* operator constructs a surface which joins only one of the edges of the surface pair. Since there are no corners being constructed in this operation, the restriction on the 'CUBIC' edge type which was imposed in *srfEdge* does not apply. In the 'QUADRATIC' case, the new surface is tangent continuous with the first surface edge named, but only continuous in position for the second one.

*singleSrfEdge*( *Srf1*, *Edge1*, *Srf2*, *Edge2*, *EdgeType*, *TangWeight* )

*Returns* <surface> Construct a surface joining two other surfaces along a single edge.

*Srf1*, *Srf2* <surface> The pair of surfaces to be joined.

*Edge1*, *Edge2*

<keyword> The edges to be joined, one of 'TOP', 'BOTTOM', 'LEFT', or 'RIGHT'.

*EdgeType* <keyword> One of 'LINEAR', 'QUADRATIC' or 'CUBIC', signifying the order of the cross-section of the edge.

*TangWeight*

<number | listOf number> If *EdgeType* is 'CUBIC', then tangent weights must be specified. For 'QUADRATIC' *EdgeType*, one tangent must be given. For 'LINEAR' *EdgeType*, this argument is ignored. The tangent weight for 'CUBIC' may be either a scalar or a lisp vector of two scalars, one for each side of the blend.

Figure 8-21 shows the results of constructing the edge surface between the two paraboloid surfaces from the last example. Part (a) shows the linear band produced by *BandSrf1*, and (b) shows the cubic band which is *BandSrf2*. The four individual edges in (c) are *BandSrf3a* through *BandSrf3d*.

```
BandSrf1 := srfEdge( ParabolaSrf1, ParabolaSrf2, 'linear, Nil );
```

```
BandSrf2 := srfEdge( ParabolaSrf1, ParabolaSrf2, 'cubic, '(0.05 0.025) );
```

```
BandSrf3a ~ = singleSrfEdge( ParabolaSrf1, 'top,
                             ParabolaSrf2, 'top, 'cubic, 0.3 )$
```

```
BandSrf3b ~ = singleSrfEdge( ParabolaSrf1, 'bottom,
```

```

                                ParabolaSrf2, 'bottom, 'cubic, 0.3 )$
BandSrf3c ^= singleSrfEdge( ParabolaSrf1, 'left,
                                ParabolaSrf2, 'left, 'cubic, 0.3 )$
BandSrf3d ^= singleSrfEdge( ParabolaSrf1, 'right,
                                ParabolaSrf2, 'right, 'cubic, 0.3 )$

```

## 8.9 Surface Refinement Operations

The surface refinement operation **sRefine** (see section 7.10 [Basic Surfaces], page 75) requires that the new knots for the refined surface be specified. This is often inconvenient for a designer who is not particularly concerned with the parametrization of surface, but with the geometric shapes. The three operations below, although they will not always be applicable, provide some alternatives for specifying refinement geometrically rather than parametrically. For example, rather than requesting six additional control points in the parametric range [0.5,1.5], one might request additional control points along the surface between X coordinate values of 6.8 and 9.5.

The **addFlex** operator adds (potential) flexibility to a surface without changing the shape of the surface in any way. The resulting surface representation will have a larger set of defining control points than the original, but will still specify the same shape. The new control points will be concentrated in the given range, although they may not all fall in that range. The first few new control points (approximately as many as the order of the surface) may be spread over a larger area, but remaining ones will fall into the specified region.

**addFlex( Srf, Axis, Min, Max, NewPts, RefCrv )**

**Returns** <surface> Refine a surface along a specific axis.  
**Srf** <surface> Surface to be refined.  
**Axis** <keyword> Coordinate axis for measurements ('X', 'Y', or 'Z').  
**Min, Max** <number> The lower and upper bound of the region, respectively.  
**NewPts** <number> Number of control points to add.  
**RefCrv** <boolean> Which boundary curve to measure against: If T, use the one with larger values in the orthogonal direction.

Figure 8-22 demonstrates the use of the **addFlex** operator to add detail to a surface, as in the following example. (The **skelWarp** operation adds "bumps" to surfaces, and is described in the section on Warping (see section 8.10.3 [Warping], page 127).

```

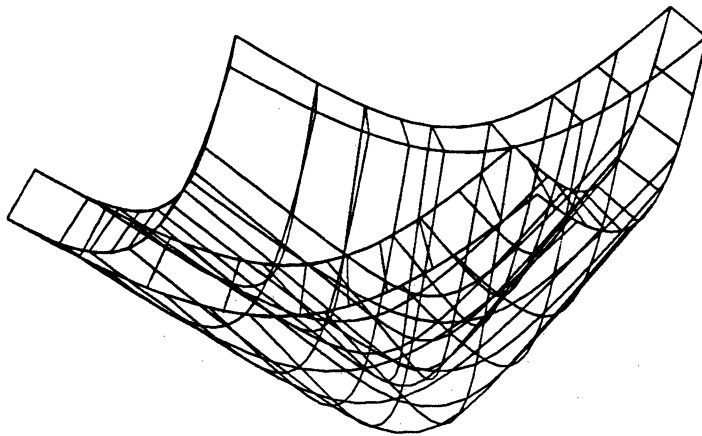
% Form a surface by making a curve and extruding it.
Crv1 := curve( parmInfo( CUBIC, EC_OPEN, KV_UNIFORM ),
               list( pt( -0.8, -0.4, 0.0 ),
                     pt( -0.4,  0.4, 0.0 ),
                     pt(  0.4,  0.4, 0.0 ),
                     pt(  0.8, -0.4, 0.0 ) ) )$

Srf1 := extrudeDir( Crv1, vec( 0, 0, -1 ) )$

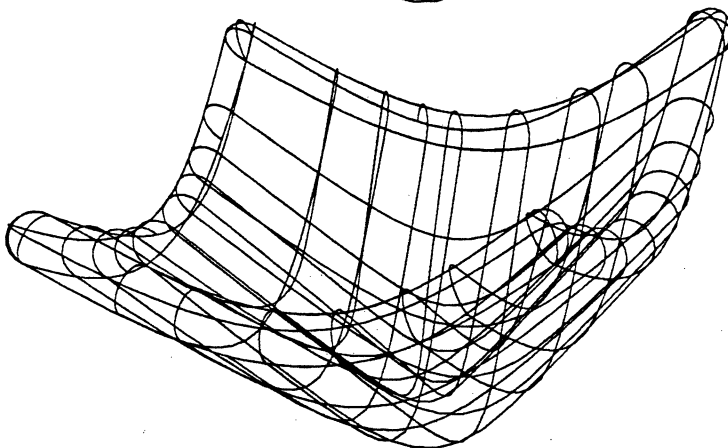
% Add extra control points to the surface, concentrated between
% x=0.0 and x=0.4.
FlexiSrf := addFlex( Srf1, 'x, 0.0, 0.4, 6, Nil )$

% Modify the surface, using the extra degrees of freedom (flexibility).
Skel := vector( pt( .21, .18, 0 ), pt( .21, .18, -1 ) );

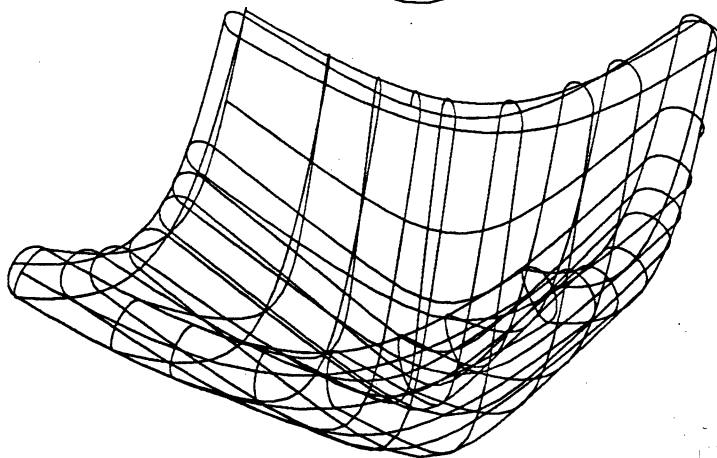
```



(a) Linear Surface Edge

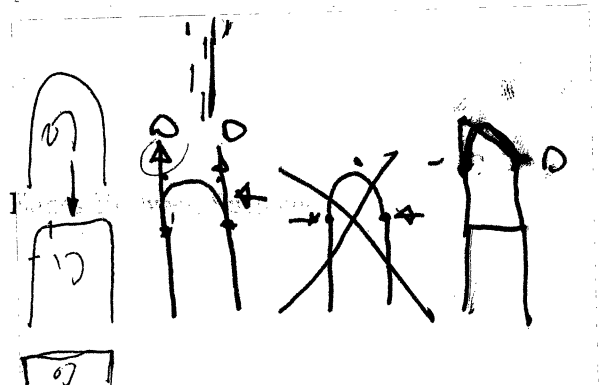


(b) Cubic Surface Edge



(c) Single Cubic Surface Edges

Figure 8-21: Examples of Constructing I



```
ModSrf := skelWarp( FlexiSrf, vec( .4, .8, 0 ),
                   0, 0, 0.5, 0.2, Nil, Skel )$
```

The **Axis** argument indicates that the region is to be determined by the X, Y, or Z coordinate of the points in the control mesh. The **Min** and **Max** arguments simply denote the bounds of the region to be refined, measured along the specified coordinate axis. The **NewPts** argument specifies (approximately) how many control points will be added to each row along the axis. These control points are spread evenly across the region.

The **addFlex** operator attempts to determine a correspondence between the coordinate axes in 3D geometric space and the coordinate axes in 2D parametric space. Determining this correspondence is simply a matter of deciding which plane (parallel to the coordinate system) the surface most nearly lies in. Two of the three geometric axes are associated with the two parametric axes. This scheme allows the user to specify regions for refinement in his normal working space — for example, based on engineering drawings which specify where a feature is to be added. It is not always possible to determine this correspondence since, of course, the surface may not be well aligned with the coordinate axes. In this case, the routine prints an error message and exits. The user should normally not have to be concerned with how this correspondence is made, but needs to be aware that it is being done since it may not always be what he expects. If the correspondence between geometric and parametric coordinate systems succeeds, there will be two (opposite) boundary curves of the surface which could both be used to measure the refinement region against. Since the results of using one or the other could be very different, the **RefCrv** flag is used to disambiguate in this case. Normally the boundary curve which has the smaller coordinate values along the axis orthogonal to the specified one is used, but if **RefCrv** is true, the other boundary curve is used.

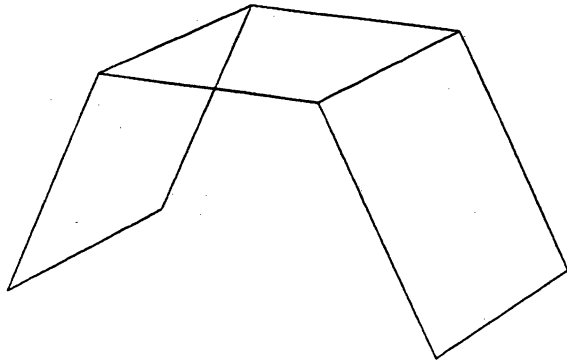
Figure 8-23(a) illustrates a surface for which the correspondence between parametric and geometric space cannot be made. Figures 8-23(b) and (c) illustrate how the **RefCrv** argument can be used to select the correct boundary for measuring along. The surface on the left used a **Nil RefCrv** argument, while the one on the right used a **T RefCrv** argument.

The **isolateRegion** operator refines a surface in such a way that later modifications to the control points inside the region will have no effect on the shape of the surface outside the region. The arguments are similar to those for the **addFlex** operator above, but the kind of refinement which is done is much different. It should be noted that this operation provides complete isolation: no continuity above position across the region boundaries is maintained by the spline once this has been done. Figure 8-24 shows the same surface modified after adding flex (a) and isolating (b) the region to be modified. Note that modifications within the region in (b) do not have any affect outside the region. **IsolateRegion** also allows sharp edges to be introduced within the surface.

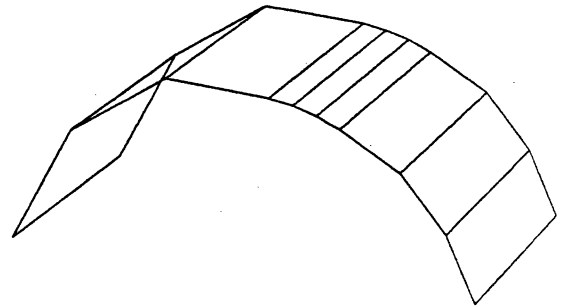
```
isolateRegion( Srf, Axis, Min, Max, RefCrv )
```

<b>Returns</b>	<surface> Refine a surface so that later modifications to a region do not affect the rest of the surface.
<b>Srf</b>	<surface> Surface to be refined.
<b>Axis</b>	<keyword> Coordinate axis for measurements, one of 'X', 'Y', 'Z'.
<b>Min, Max</b>	<number> The lower and upper bound of the region, respectively.
<b>RefCrv</b>	<boolean> Which boundary curve to measure against; if true use the one with larger values in the orthogonal direction.

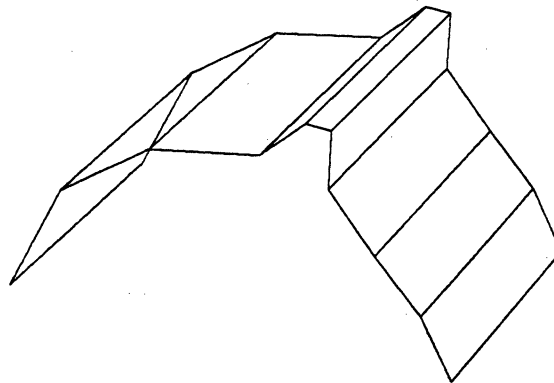
Finally, one other kind of refinement is available: the introduction of an isoparametric line of discontinuity across a surface. The **featureLine** function adds multiple knots at the location given using the same geometric space specification as for **addFlex** and **isolateRegion** above. The **ContinuityClass** argument specifies how much continuity is retained across the feature line. An argument of 0 indicates that not even positional continuity is maintained, 1 keeps positional continuity, 2 will



(a) Original Mesh

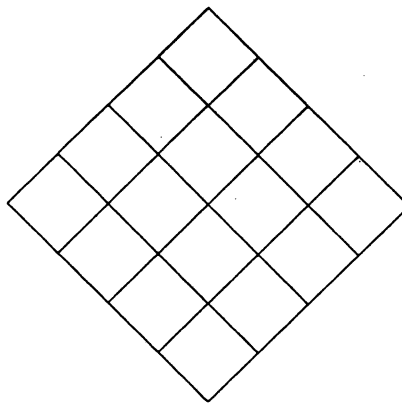


(b) Mesh With Flex Added

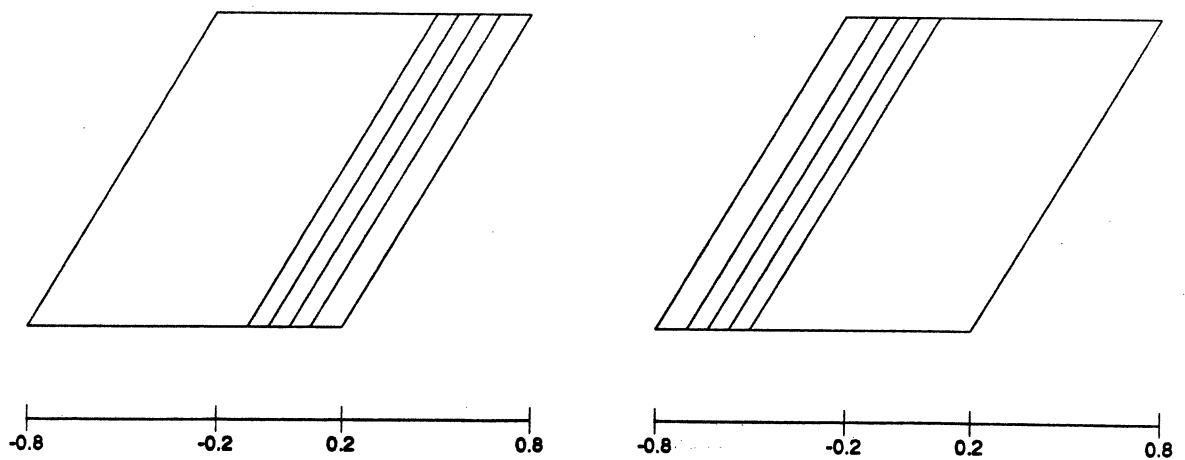


(c) Mesh Modified

**Figure 8-22: Adding Flexibility to a Surface**



(a) Parametric correspondence undetermined

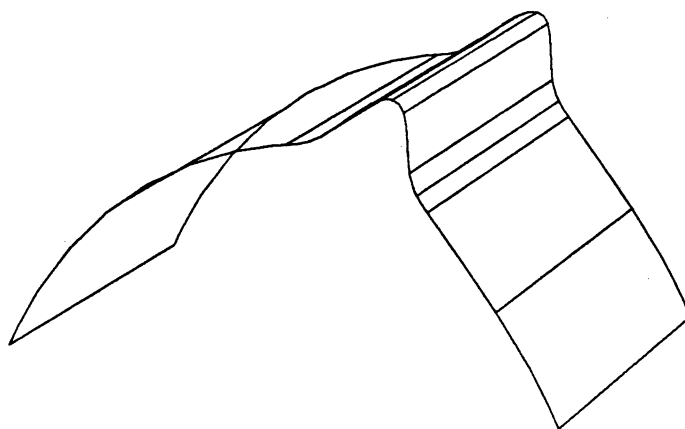


(b) Range  $[-0.1, 0.1]$ , RefCrv = nil

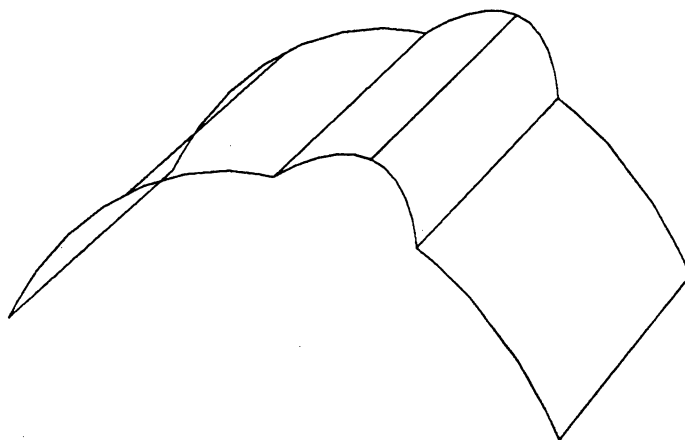
(c) Range  $[-0.1, 0.1]$ , RefCrv = T

Figure 8-23: Parametric Correspondence for addFlex Operator





(a) Modified After AddFlex



(b) Modified After IsolateRegion

**Figure 8-24:** Comparison of **addFlex** and **isolateRegion** Operations

keep tangent continuity, etc. If a continuity class greater than the surface order minus 1 is specified, then no action is taken. (e.g., for cubic curves, where order = 4, the largest value that makes sense is 3). Values of 0 and 1 will be the most useful because these produce control points along the feature line that can be manipulated to define the feature exactly.

**featureLine( Srf, Axis, Value, ContinuityClass, RefCrv )**

**Returns** <surface> Refine a surface so that a crease can be inserted.  
**Srf** <surface> The surface to be refined.  
**Axis** <keyword> Coordinate axis for measurements, one of 'X', 'Y', 'Z'.  
**Value** <number> The coordinate along Axis at which the feature line is to be added.  
**ContinuityClass** <number> How much continuity should be maintained across the feature line.  
**RefCrv** <boolean> Which boundary curve to measure against; if true, the one with larger values along the orthogonal coordinate axis is used.

## 8.10 Surface Modification Operations

Up to this point, the surface operations have either constructed surfaces or refined them without changing the shape of the surface. The operators described in this section are designed to modify the shapes of surfaces in clear and definite ways. Many of the operations can also be applied to curves, polylines, or other objects containing points. These operations list the type of the first argument as <object> instead of <surface> to indicate the generality.

### 8.10.1 Bending

The bend operator performs an idealized bending (no physical stretching or compressing of the material is calculated) of the given object according to the chosen parameters.

**bend( Obj, Axis, Angle, Center, Min, Max, LH )**

**Returns** <surface> Bend a surface along an axis with a specified angle and spread.  
**Obj** <object | listOf object | vectorOf object> Surface to be bent (or list of surfaces).  
**Axis** <keyword> Axis along which the bend is to take place ('X', 'Y', or 'Z').  
**Angle** <number> Bending angle, measured in degrees.  
**Center** <number> Coordinate along the specified axis which is the center of the bend.  
**Min, Max** <number> Coordinate along the given axis which specifies the lower (or upper) end of the bending range.  
**LH** <boolean> Indicates which axis to bend towards. If Nil, a right-handed choice is made. If T, a left-handed system is used.

One can think of the **bend** operator as modifying the specified axis. If X is specified and the surface is a rod lying along the X axis, then the rod will be bent in the usual way. There are two possible directions to bend the X axis: towards Y or towards Z. The default (unless the **LH** flag is true) is to bend towards the next (right-handed) axis. In the case of X, that would be bending towards Y. However, if the **LH** flag is true, the bend will go towards Z.

The *Angle* determines the total angle which the bend will occupy. The bending range specifies the distance and position along the given axis over which the bend will be spread. A very short bending range will produce a sharp bend, while longer ranges will produce more gradual bends. Finally, *Center* determines the point of tangency of the bent surface to the original surface — or the fixed point of the bend. If the bending center is in the middle of the bending range, then the surface will be bent up on either side, leaving the surface tangent to the original surface at the center point. If the bending center is at the left end of the range, then the right side of the surface will be bent up, leaving the bending center (and all points to the left) tangent to the original surface. Figure 8-25 shows bending a curve to illustrate various values of the bending parameters.

### 8.10.2 Derived From Linear Transformations

A stretching operator is available which is primarily useful for adjusting the proportions of surfaces. It is no different than the graphical scaling transformations, but is presented here in the modeling context for ease of use.

**stretch**( *Obj*, *XScale*, *YScale*, *ZScale* )

*Returns*     <surface> Construct a surface by stretching a reference surface by specified amounts over each axis.

*Obj*           <object | listOf object | vectorOf object> Surface to be stretched (or list of surfaces).

*XScale*, *YScale*, *ZScale*

              <number> Amount of stretch along each respective axis.

The **taper** operator produces a surface which is tapered along the specified axis according to the values returned by a user-provided tapering function. One can think of the taper as a variable stretch relative to a single axis, with the scaling values coming from the user-provided function. That function should map values on the specified coordinate axis to scale factors which specify how much to scale one of the other coordinates. Only one of the other coordinates is scaled. If you specify tapering relative to the Y axis, the Z coordinates normally get scaled. If *LH* is true, however, the X coordinates will get scaled instead. For a circularly symmetric taper about the axis, apply the **taper** function twice with the *LH* argument set to true for one of them. Figure 8-26 shows the result of tapering a flat surface along the X axis.

**taper**( *Obj*, *Axis*, *LH*, *TaperFn*, *ExtraArgs*, ... )

*Returns*     <surface> Construct a surface by “tapering” a reference surface.

*Obj*           <object | listOf object | vectorOf object> Surface to be tapered (or list of surfaces).

*Axis*         <keyword> Axis (coordinate) used to control the tapering, given as 'X', 'Y' or 'Z'.

*LH*            <boolean> Indicates which axis (coordinate) to scale.

*TaperFn*     <function> User provided function which is used to determine the shape of the taper.

*ExtraArgs*, ...

              <anything> List of extra arguments to *TaperFn*.

% Use a rectangular, flat initial surface.

```
Srf := flatSrfBounds( 4, 4,                                % Cubic
                     5, 12,                               % Plenty of control points.
                     pt( -0.7, -0.4 ),                   % Lower left corner
```

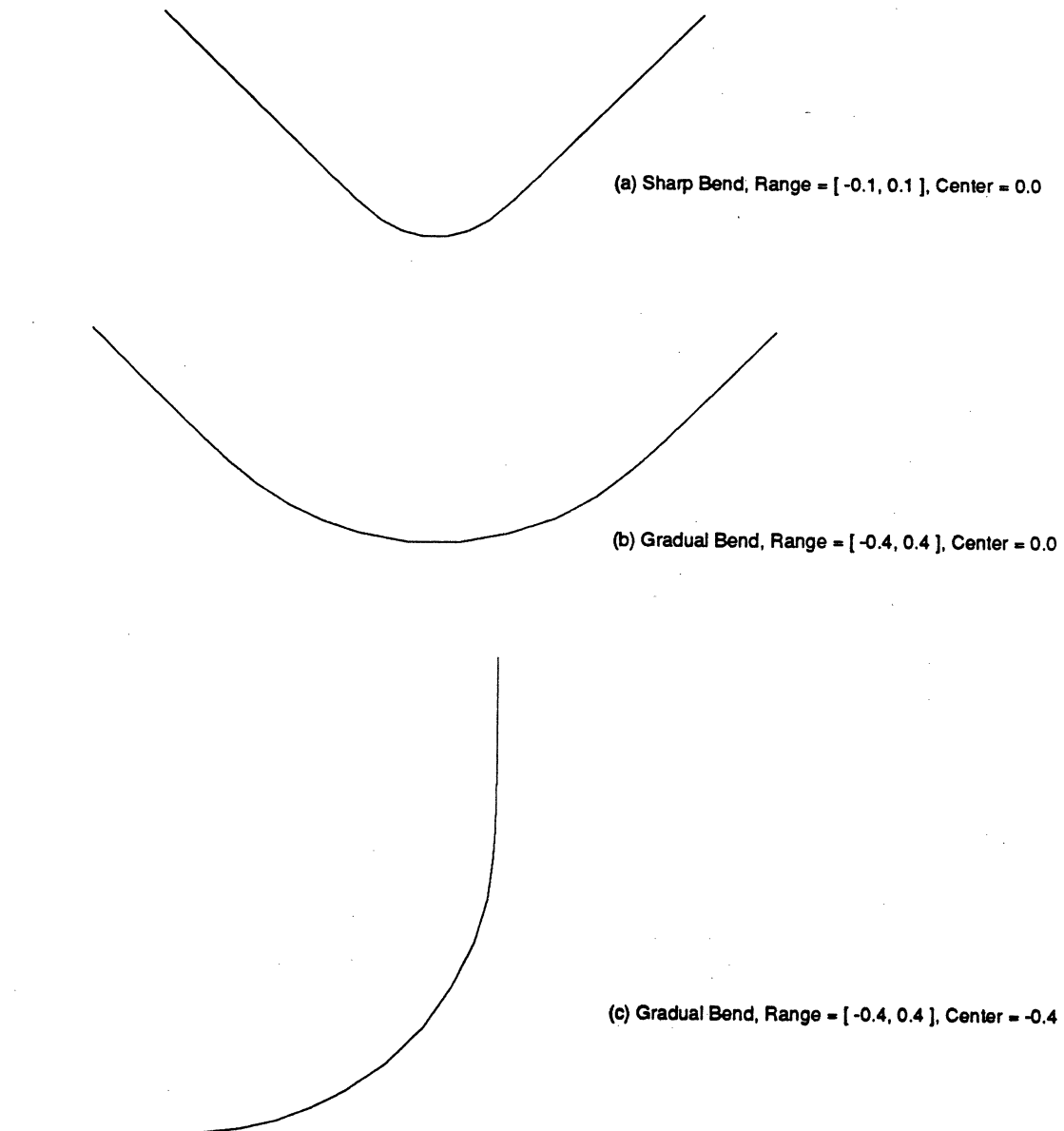


Figure 8-25: Various Bending Parameters Applied to a Straight Line

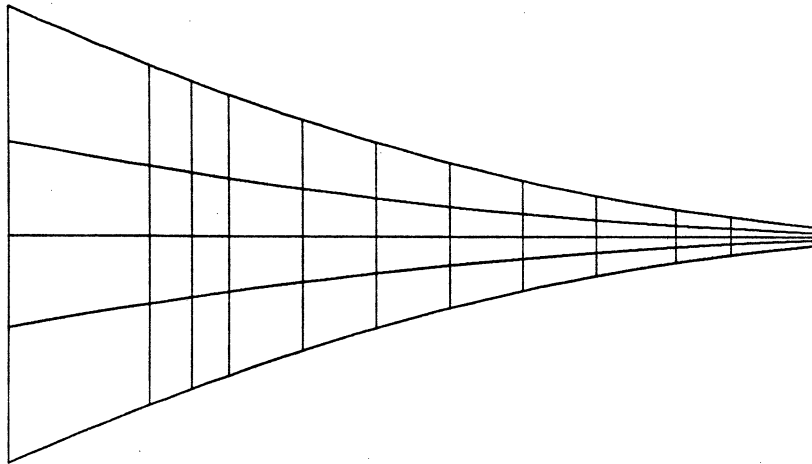


Figure 8-26: Example of taper Operator

```

                                pt( 0.7, 0.4 ) );    % Upper right corner

% Define a mapping of x coordinates.
procedure taperFn( x );
begin
    scalar Xtmp;

    % Map range [ -.7, .7 ] to [ 1, .1 ] quadratically
    Xtmp := 1.0 - (0.8/1.4) * (x+0.7);
    return Xtmp*Xtmp;
end;

% Apply the quadratic tapering to the rectangular surface.
TaperSrf ^= taper( Srf, 'x, Nil, 'taperFn, Nil )$

```

Just as the **taper** operator can be thought of as a variable scaling operation, the **twist** operator can be thought of as a variable rotation. Instead of rotating each point about an axis by a constant amount, the angle of rotation is determined for each point by the coordinate value along the specified axis. The user-provided twisting routine maps that value into a rotation angle in radians.

```
twist( Obj, Axis, TwistFn, ExtraArgs, ... )
```

**Returns**     <surface> Construct a surface by performing a variable rotation of another surface.

**Obj**           <object | listOf object | vectorOf object> Surface to be twisted (or list of surfaces).

**Axis** <keyword> Twist axis, points are rotated about this axis based on the corresponding coordinate of the point. Given as 'X', 'Y', or 'Z'.  
**TwistFn** <function> Determines the rotation angle (in radians) for each point as a function of the given axis coordinate.  
**ExtraArgs, ...** <anything> List of extra arguments to be passed to *TwistFn*.

There are variants of **taper** and **twist** which allow a special spline curves to be used for the twisting and tapering control functions. These are called **taperWithSplineFn** and **twistWithSplineFn**. The spline curve arguments are 2D, where the Y coordinates are interpreted as a function of X. For the most intuitive results, the X values should occur at the nodes of the spline. There are four functions provided for creating these curves, and they insure that those conditions hold.

**taperWithSplineFn**( *Obj*, *Axis*, *LH*, *TaperFn* )

**Returns** <surface> Construct a surface by tapering a reference surface.  
**Obj** <surface> The surface to be tapered.  
**Axis** <keyword> Axis (coordinate) used to control the tapering, given as 'X', 'Y' or 'Z'.  
**LH** <boolean> Indicates which axis (coordinate) to scale.  
**TaperFn** <curve> The spline curve which describes the tapering.

**twistWithSplineFn**( *Obj*, *Axis*, *TwistFn* )

**Returns** <surface> Construct a surface by performing a variable rotation of another surface.  
**Obj** <object | listOf object | vectorOf object> Surface to be twisted (or list of surfaces).  
**Axis** <keyword> Twist axis, points are rotated about this axis based on the corresponding coordinate of the point. Given as 'X', 'Y', or 'Z'.  
**TwistFn** <curve> The curve which controls the amount of twisting.

**constantSplineFn**( *BeginParm*, *EndParm*, *Val* )

**Returns** <curve> Construct a constant spline curve for use as a function f(t).  
**BeginParm** <number> Parameter value at the beginning of the curve.  
**EndParm** <number> Parameter value at the end of the curve.  
**Val** <number> The constant value of the curve.

**linearSplineFn**( *BeginParm*, *EndParm*, *BeginVal*, *EndVal* )

**Returns** <curve> Construct a linear spline curve for use as a function f(t).  
**BeginParm** <number> Parameter value at the beginning of the curve.  
**EndParm** <number> Parameter value at the end of the curve.  
**BeginVal** <number> Value of the curve at the beginning.  
**EndVal** <number> Value of the curve at the end.

**approxValuesSplineFn**( *BegParm*, *EndParm*, *ValList*, *UseY* )

**Returns** <curve> Construct a spline curve for use as a function f(t) by approximating a set of values.  
**BegParm** <number> Parameter value at the beginning of the curve.

**EndParm** <number> Parameter value at the end of the curve.  
**ValList** <listOf number | listOf euclidPoint> The list of values for the function to approximate. If the list is points, the X coordinates will be extracted (unless *UseY* is true) and used as the values.  
**UseY** <boolean> If *ValList* contains points, and this flag is not Nil, the Y coordinates will be used as values rather than X.  
**interpValuesSplineFn( BeginParm, EndParm, ValList, FnY )**  
**Returns** <curve> Construct a spline curve for use as a function  $f(t)$  by interpolating a set of values.  
**BegParm** <number> Parameter value at the beginning of the curve.  
**EndParm** <number> Parameter value at the end of the curve.  
**ValList** <listOf number | listOf euclidPoint> The list of values for the function to interpolate. If the list is points, the X coordinates will be extracted (unless *FnY* is true) and used as the values.  
**FnY** <boolean> If *ValList* contains points, and this flag is not Nil, the Y coordinates will be used as values rather than X.

### 8.10.3 Warping

#### Basic Warp

The basic warping operator puts smooth, usually round, bumps onto surfaces.

**warp( Obj, Direction, Center, WFactor, Cutoff, MInfo )**  
**Returns** <surface> Construct a surface by "warping" another surface.  
**Obj** <object | listOf object | vectorOf object> Surface to be warped.  
**Direction** <geomVector> Direction of warp.  
**Center** <point> The point which is to be the center of the warp.  
**WFactor** <float> A real number which describes how sharply the warp falls off with distance.  
**Cutoff** <number> Distance at which the warp effect should go to zero (roughly the radius of the bump).  
**MInfo** <number> Specification of how distances are to be measured (described below) (default 2).

The warp operator modifies a surface by moving control points in a specified direction. The magnitude of the direction vector is the maximum distance which a control point may be moved. The *Center* argument specifies the point which is to be the center of the warp. This point, if it were a control point of the mesh, would be moved the entire distance specified by the magnitude of the *Direction*. Other points in the "neighborhood" of the specified point will be moved also. The amount which they are moved depends on their distance from the center of the warp, the warp factor, and the specified cutoff distance.

The *MInfo* argument consists of a number which describes which distance measure to use. The default is 2, for the standard euclidean (L2) distance measure. A value of *N* says to use the LN norm to compute distance. The LN norm is the Nth root of the sum of the Nth power of each coordinate. If *MInfo* is not specified (Nil), then the default (2) is used. Various values of *MInfo* can be used to adjust the shape of the warp to some degree. *MInfo* = 2 gives round bumps, while for larger values the bump becomes more like a rectangle aligned with the coordinate axes. The

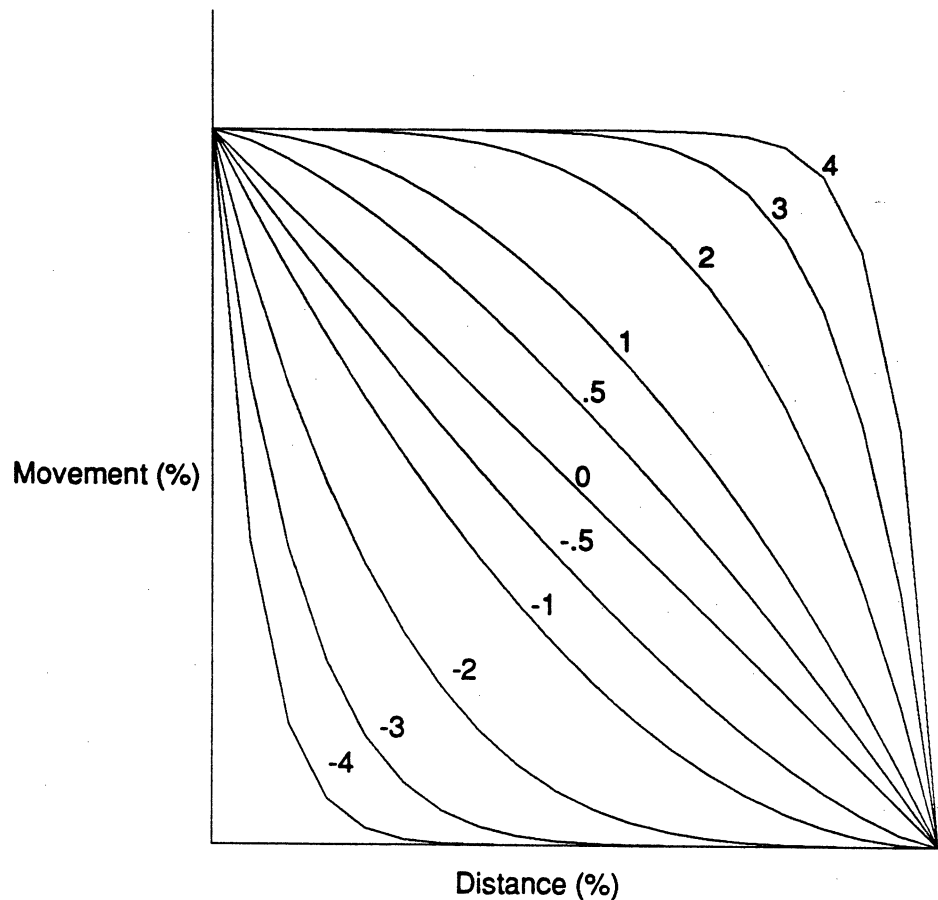


Figure 8-27: Graphs of the Effect of the Warp Factor

larger the value, the sharper the rectangular corners of the bump.  $MInfo = 1$  gives rectangularly shaped bumps too, but they are aligned at a diagonal to the coordinate axes.

To get an idea of how the  $WFactor$  affects the warping, see Figure 8-27 which diagrams the percentage of the maximum point movement as a function of distance relative to the cutoff distance.

Some basic guidelines for selecting the  $WFactor$  are:

1. A value of 0 is a simple linear warp — the function linearly goes from 1 to 0 as the distance from the warp center goes from 0 to the cutoff distance.
2. Increasingly negative numbers make the function drop off more rapidly. A value of -8.0 is almost a delta function; the only point that will be moved significantly is the one at the warp center. Values of -1.0 to -2.0 produce gradual dropoffs with more weight towards the central points.
3. Increasingly positive numbers make the function drop off more rapidly, but only when it is almost at the maximum distance. A value of +8.0 is almost a box function; this time almost all of the points will be moved the maximum distance as long as they fall within the cutoff distance. Values of 1.0 to 2.0 produce slightly more gradual dropoffs.



Figure 8-28 shows the effect of various warp factors for each of several distance measures in the weighting phase.

A variant of **warp**, called **warpR**, allows an extra restriction list argument which restricts the operation to certain regions. The restriction list will be described in the section on flattening (see section 8.10.4 [Flattening], page 132).

**warpR**( *Obj*, *Direction*, *Center*, *WFactor*, *Cutoff*, *MInfo*, *Restrict* )

**Returns** <surface> A variant of **warp** which constructs a warped surface with the warp restricted to certain regions.

**Obj** <object | listOf object | vectorOf object> The object to warp.

**Direction** <geomVector> The warp direction.

**Center** <point> The point at the center of the warp.

**WFactor** <float> A factor which determines how sharply the warp falls off with distance.

**Cutoff** <number> Distance at which the warp effect should go to zero (roughly the radius of the bump).

**MInfo** <number> Specification of how distances are to be measured (described above).

**Restrict** <restriction> A list describing the bounds of the warp. See section 8.10.4 [Flattening], page 132 for more information.

### Skeletal Warp

An extension of the basic warp is to allow the shape of the warp to be determined more exactly. Two operators are provided: **skelWarp**, which makes the warp according to some polyline shape, and **regionWarp** which warps according to a closed polygonal region. We first describe the skeletal warp (the polyline shape is referred to as the *skeleton* of the warp).

**skelWarp**( *Obj*, *Direction*, *WFactor*, *Cutoff*, *MInfo*, *Skel* )

**Returns** <surface> Warp an outline on a surface using a polyline as a template.

**Obj** <object | listOf object | vectorOf object> Surface to be warped (or list of surfaces).

**Direction** <geomVector> Direction of the warp.

**WFactor** <float> A real number which describes how sharply the warp falls off with distance.

**Cutoff** <number> Distance at which warp effect should go to zero. In this case, this is roughly half the cross-sectional size of the warp.

**MInfo** <number> Information about the distance measurements.

**Skel** <polyline | polygon | curve | listOf point | vectorOf point> Skeleton from which to measure. May also be a polyline, or a curve (in which case the control polygon is used as the skeleton).

The distance is measured from the set of points provided as the skeleton. (The points can be thought of as the defining data for a polyline.) The *MInfo* is used as in the simple **warp** operation.

Whereas the "bumps" imposed by the standard warp are basically square or circular in shape, skeletal warps can resemble long lines or curves of a particular width. While the skeleton must be given as a polyline, any desired curve can be approximated as a polyline by doing an appropriate refinement and then extracting the control polygon. For intuitive results, the skeleton should probably lie close to or in the surface. A simple example of a skeletal warp was given with the description of **addFlex** in the section on Surface Refinement Operations (see section 8.9 [Surface

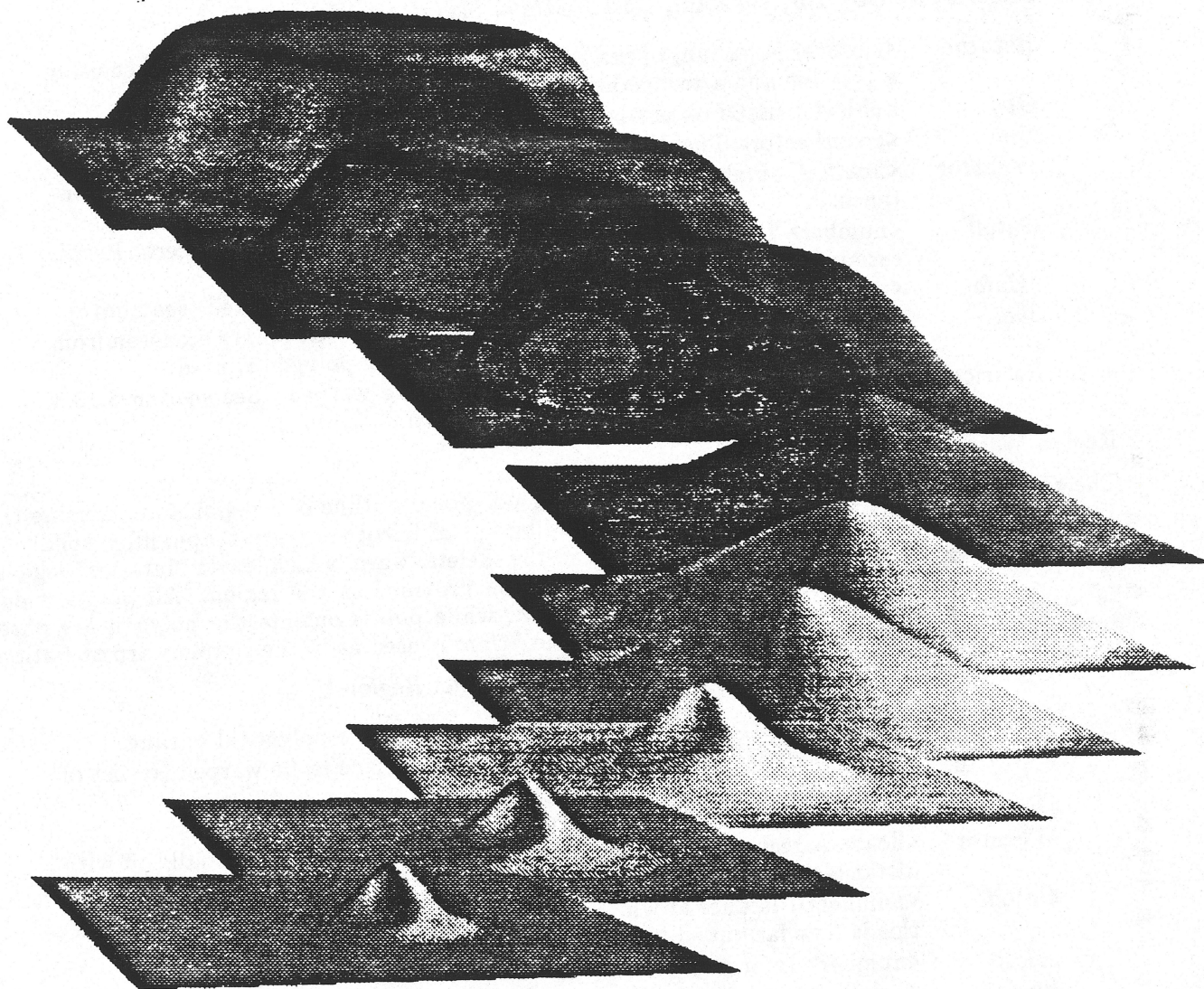


Figure 8-28: Varying Warp Factors and Distance Measures Applied to a Flat Surface

Refinement Operations], page 116.

Like **warp**, a variant operation, **skelWarpR** allows an extra argument to restrict the regions to which the warp will be applied:

**skelWarpR**( *Obj*, *Dir*, *WFactor*, *Cutoff*, *MInfo*, *Skel*, *Restrict* )

**Returns** <surface> A variant of **skelWarp** which warps an outline on a surface using a polyline and a restriction list.

**Obj** <object | listOf object | vectorOf object> The object to warp.

**Dir** <geomVector> The direction of the warp.

**WFactor** <float> A number which describes how sharply the warp falls off with distance.

**Cutoff** <number> The distance at which the warp effect should go to zero. In this case it is roughly half the cross-sectional size of the warp.

**MInfo** <number> Specification of how distances are to be measured (see above).

**Skel** <polyline | polygon | curve | listOf point | vectorOf point> Skeleton from which to measure. If this is a curve the control polygon is used.

**Restrict** <restriction> A restriction on the area to be warped. See section 8.10.4 [Flattening], page 132 for more information.

### Region Warp

The **regionWarp** variation of the basic warp uses a polygonal outline of a region to determine the approximate shape of the warp. This warp allows bumps of fairly arbitrary shape, filled solidly in the interior. Note the difference between this and the skeletal warp, which leaves "interior" regions as they were. Distance is measured from the polygon provided as the region. All points which lie inside the polygon are offset by the full amount, while points outside the polygon are offset depending on their distance from the polygon. The *MInfo* is used as in the simple warp operation.

**regionWarp**( *Obj*, *Direction*, *WFactor*, *Cutoff*, *MInfo*, *Region* )

**Returns** <surface> Warp a region on a surface defined by a polygonal outline.

**Obj** <object | listOf object | vectorOf object> Surface to be warped (or list of surfaces).

**Direction** <geomVector> Direction of warp.

**WFactor** <float> A real number which describes how sharply the warp falls off with distance.

**Cutoff** <number> Distance at which the warp effect should go to zero. In this case, this is how far outside the region boundary points should be affected.

**MInfo** <number> Information about the distance measurements.

**Region** <polyline | polygon | curve | listOf point | vectorOf point> Polyline describing outline of warp shape. Must be in the XY-plane.

The **regionWarp** function also allows the restriction list variant:

**regionWarpR**( *Obj*, *Dir*, *WFactor*, *Cutoff*, *MInfo*, *Region*, *Restrict* )

**Returns** <surface> Warp a region on a surface using a polygonal outline and a restriction list.

**Obj** <object | listOf object | vectorOf object> The object to warp.

**Dir** <geomVector> The direction of the warp.

**WFactor** <float> A number which describes how sharply the warp falls off with distance.

**Cutoff** <number> The distance at which the warp should go to zero. In this case, it is how far outside the region boundary points should be affected.

**MInfo** <number> Information about the distance measure to be used.

**Region** <polyline | polygon | curve | listOf point | vectorOf point> The outline of the region to be warped.

**Restrict** <restriction> A restriction of the region to be warped. See section 8.10.4 [Flattening], page 132 for more information.

Figure 8-29 shows a surface which has had a **regionWarp** applied.

```
% Note: construction of the profileCrv is not included in this
% example.

KybdSrf ~= extrudeDir( profileCrv, vec( 0, 0, -20.0 ) )$
KybdSrf ~= raiseSurface( KybdSrf, COL, Cubic )$

% Rotate it so that looking down on the XY-plane is a top view.
KybdSrf ~= objTransform( KybdSrf, ry( -90 ), rx( 90 ) )$

% Now apply the warp.
KybdPts := list( pt( 0.3, -1.3, 0.0 ),
                  pt( 20.0 - 0.3, -1.3, 0.0 ),
                  pt( 20.0 - 0.3, -7.0 + 0.3, 0.0 ),
                  pt( 0.3, -7.0 + 0.3, 0.0 ) );

% Refine the keyboard in the areas where it's needed for control
% over the resulting warp.
KybdSrf := addFlex( KybdSrf, 'X', 0.1, 0.5, 4, Nil )$
KybdSrf := addFlex( KybdSrf, 'X', 19.5, 19.8, 4, Nil )$
KybdSrf := addFlex( KybdSrf, 'Y', -1.2, -1.4, 3, Nil )$
KybdSrf ~= addFlex( KybdSrf, 'Y', -6.6, -6.8, 3, Nil )$

RegWarpSrf ~= regionWarp( KybdSrf, vec( 0, 0, -0.2 ),
                          1.0, 0.01, Nil, KybdPts )$
```

## 8.10.4 Flattening

The **flattenSrf** operator is conceptually (but not generally in practice) an inverse of the warping operator. Where **warp** can add bumps to flat surfaces, the **flattenSrf** operator can introduce flat areas in bumpy or rounded surfaces. All points on the positive side of the plane are projected onto the plane in the given direction. This has the effect of flattening the surface as if the plane was a large flat plate pushing on it. If **OtherSide** is true, then only points on the negative side are projected. If **Dir** is given as Nil, then the points are projected onto the plane in a direction normal to the plane.

**flattenSrf**( *Obj*, *Plane*, *Dir*, *OtherSide* )

**Returns** <surface> Create flat areas on a surface.

**Obj** <object | listOf object | vectorOf object> Surface to be flattened (or list of surfaces).

**Plane** <plane> Plane of the flattened area.

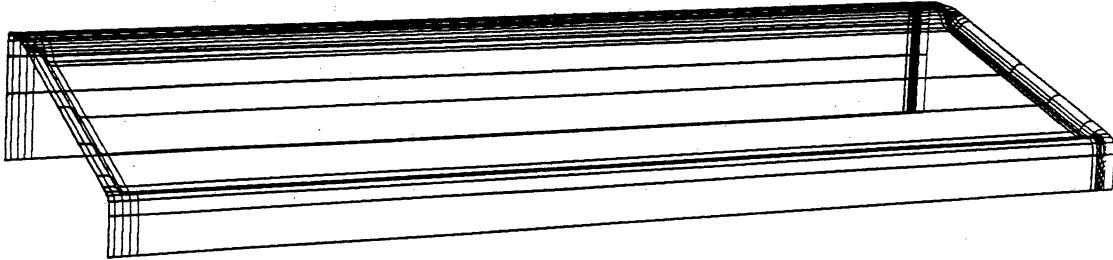


Figure 8-29: Example of regionWarp Operation

**Dir** <geomVector | Nil> Direction to project affected points to the plane (default is normal to *Plane*).

**OtherSide** <boolean> If true, only project points on the negative side of the plane.

An alternative version, **flattenSrfR**, has an extra restriction list argument at the end. The restriction list is an experimental option which may later be added to some of the other surface operations. Currently, it is available only for **flattenSrf** and the warping functions (**warpR**, **skelWarpR**, **regionWarpR**). It is almost required in order to make the **flattenSrf** operator useful, but is somewhat cumbersome to use in the initial experimental version. New users of **flattenSrf** should probably try to see how far they can get with the basic **flattenSrf**, but may find that they soon need the extra power of specifying what is to be affected via **flattenSrfR**.

**flattenSrfR**( *Obj*, *Plane*, *Dir*, *OtherSide*, *Restrict* )

**Returns** <surface> Produce flat regions on surfaces using a restriction list.

**Obj** <object | listOf object | vectorOf object> The object to flatten.

**Plane** <plane> The plane of the flattened area.

**Dir** <geomVector | Nil> The direction to project affected points to the plane (default is normal to *Plane*).

**OtherSide** <boolean> If true, only project points which are on the negative side of the plane.

**Restrict** <restriction> A restriction on the area to be flattened.

### Restriction Lists

The user sets up the restrictions which are to be applied before calling **flattenSrfR** or one of the **warpR** routines. In this way, a single restriction set could be used by several operations. The restriction expression is constructed by invoking **buildRestriction**, where the *RestrictionSpecification* is a list of restrictions to be imposed as described below. The function processes the user form into an internal form which is simple for the **flattenSrfR** routine to use. Within **flattenSrfR**, each point is tested against the restriction expression. If the expression excludes it, then the point is not considered as a candidate for being projected, regardless of which side of the plane it lies on. Similarly, in the **warpR** routines, a point is only considered as a candidate for being moved if it



is not excluded by the restriction expression. The **buildRestriction** routine returns a list of the restrictions. A complex set can be built incrementally using **addRestriction**.

**buildRestriction**( *RestrictionSpecification1*, ... )

**Returns** <restriction> Build a restriction expression for use with the **warpR**, **skelWarpR**, **regionWarpR**, and **flattenSrfR** operators.

**RestrictionSpecification**

<restriction> A description of the area enclosed by the restriction. See below for more information.

**addRestriction**( *OldRestrictions*, *Restrict1*, ... )

**Returns** <restriction> Add a restriction to an existing restriction list.

**OldRestrictions**

<restriction> The original restriction list.

**Restrict1** <restriction> A description of the area enclosed by the restriction. See below for more information.

The restrictions that can be specified are:

**rInside**( *R*, *ViewProjection* )

A point must be inside the region *R* (a polygon) when projected into the specified view. *R* is projected the same way. The view projection is determined as follows:

'XY            (x,y,z) -> ( x, y )

'YZ            (x,y,z) -> ( y, z )

'XZ            (x,y,z) -> ( x, z )

**rOutside**( *R*, *ViewProjection* )

As above, but the point is required to be outside the region *R*.

**rPositiveSide**( *Plane* )

Point must be on positive side of the given plane (a different plane than the one onto which the surface is being flattened).

**rNegativeSide**( *Plane* )

As above, but point is on negative side of plane.

**rLessThan**( *Coord*, *CoordVal* )

*Coord* is 'X', 'Y', or 'Z'. The corresponding coordinate of the point must be less than *CoordVal*. Note that this is just a shorthand (and more efficient way) of specifying the two plane restrictions above in cases when the plane is parallel to an axis.

**rGreaterThan**( *Coord*, *CoordVal* )

As above, except point value must be greater than *CoordVal*.

You can check that a restriction list has the desired effect on a sample point using **checkRestrictions**, which returns a boolean value indicating whether the point is excluded by the given restrictions.

**checkRestrictions**( *Pt*, *RestrictionList* )

**Returns** <boolean> True if the given pint is included in the region described by the given restriction list.

**Pt** <point> The point to test.

**RestrictionList**

<restriction> The restriction list to check.

Figure 8-30 shows the use of a restriction list to achieve a desired result on a surface (b) which was not possible using the simpler form of the flatten operation (a).

```
TestSrf := flatSrf( 4, 4, 8, 8 )$

% This function will approximate a paraboloid.
procedure parabFn( S, Index1, Index2 );
    distPtPt( S->sMesh[Index1][Index2], Origin );

Dir := vec( 0, 0, 1 );
ParabolaSrf1 := lift( TestSrf, Dir, 'parabFn )$

Pln := planeThruPtWithNormal( pt( 0.0, 0.0, 0.75 ), vec( 0, 0, 1 ) );
FSrf1 ^= flattenSrf( ParabolaSrf1, Pln, Nil, T, Nil )$
FSrf2 ^= flattenSrfR( ParabolaSrf1, Pln, Nil, T,
    buildRestriction( 'x, 0.0 ) )$
```

### 8.10.5 Curve-Based Modification

The operator discussed in this section modifies each "curve" in a surface to produce a modified surface as a result. These "curves" do not actually lie in the surface, but are the curves defined by the rows or columns of the control mesh (see section 7.10 [Basic Surfaces], page 75). They are the blending curves described earlier in this manual (see chapter 3 [Spline Introduction], page 11). This operator may be useful for applications programmers to customize to produce specialized surface operators; it is somewhat complex to be employed extensively from the user level.

The *loftHit* operation uses the original surface to create a new surface. The new surface is constructed by applying *GeomFn* to each row (or column, given by *Dir*) curve from the original surface (extracted using *crvFromSrf*). The new surface inherits one knot vector from the original surface, and the other is determined from the row or column curves resulting from the application of the *GeomFn*.

```
loftHit( Srf, Dir, GeomFn, ExtraArgs )
```

**Returns** <surface> Construct a surface by applying a user defined function to each row or column of a control mesh.

**Srf** <surface> Surface to be operated on.

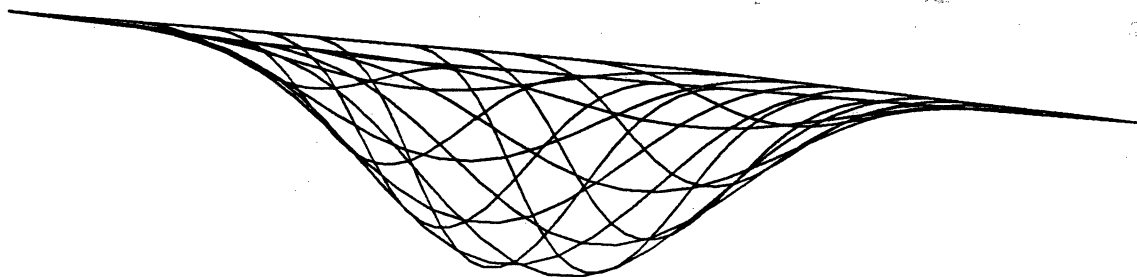
**Dir** <keyword> Direction of operation, one of 'ROW or 'COLUMN.

**GeomFn** <function> Function applied to each curve.

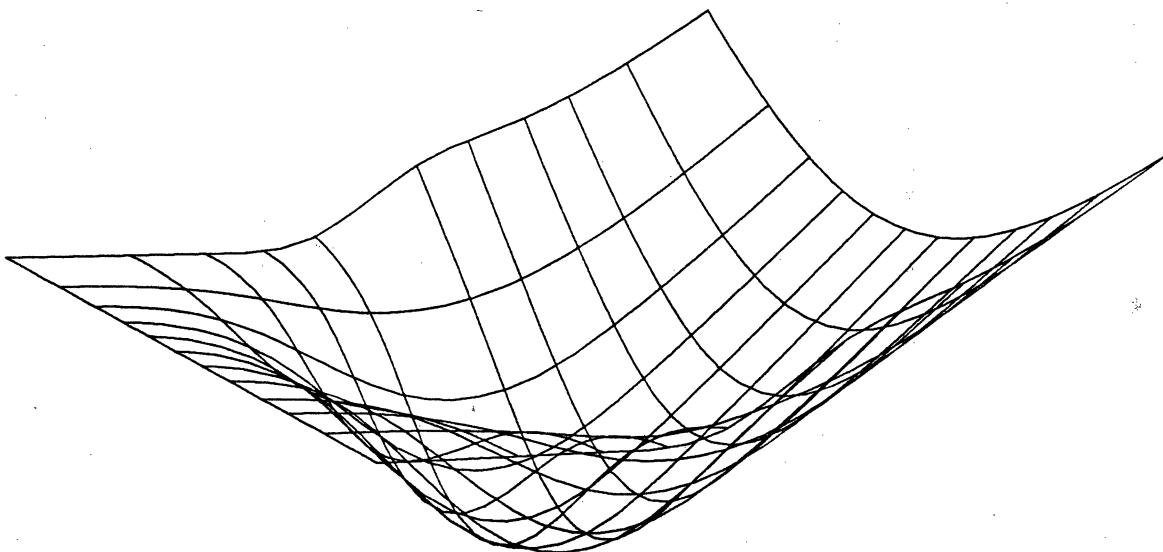
**ExtraArgs** <anything> Extra arguments for the *GeomFn*.

In using this routine, one part which requires special consideration is providing the *GeomFn*, which must be carefully written if it is to work on a fairly general class of surfaces. The *GeomFn* may frequently have to deal with the control points in a bare knuckles approach since the curve modification operators are less well-developed in the system than the surface operators, although a few higher level curve modification tools will help in this task.

The *GeomFn* function is assumed to take at least two arguments: the curve which is to be modified and the index of that curve in the original surface mesh. The extra arguments supplied by the caller are added at the end of these two required arguments.



(a) Simple Flatten Operation



(b) Flatten Operation with Restriction

**Figure 8-30:** Example of **flatten** Operation With and Without Restriction List



## 8.11 Curve Modification Operations

Curve operators are somewhat limited in the current system, because sculptured surface design has been emphasized over curve design. The following small set of operators, mostly adapted for curves from surface analogues, may be helpful. In addition, recall that most of the surface modification operations also work for curves or other objects containing point data.

The **crvTangLines** function constructs a cubic curve which is tangent to a given set of lines, in the order given. This allows one to do something a little smoother than piecing together a set of arcs, which is often frustrating when trying to build "aesthetic" curves.

**crvTangLines**( *LineList* )

*Returns*     <curve> Construct a curve tangent to a set of lines.

*LineList*    <list> List of lines and tangency points, as described below.

Each *LineList* element is a list consisting of three elements:

( *Dir Intersect Tightness* )

*Dir*            The line the curve must be tangent to. Alternatively, it may be a vector giving the direction of the tangent line.

*Intersect*    Another line which determines the point of tangency by its intersection with the first line. Alternatively, it may be a point which is the tangency point. If *Dir* is a vector, *Intersect* must be a point.

*Tightness*    Describes how closely the curve will follow the first line. Given as a floating point number in the range [0,1]. Larger values indicate that the curve follows the line quite closely in the neighborhood. The extreme values are either end of the range are best avoided.

The **crvTangLines** operator returns a cubic piecewise Bezier curve. During further operations, however, it is sometimes difficult to maintain the continuity which was produced. An alternate routine, **crvTangLines2**, looks exactly the same from the outside but returns a cubic curve which does not have potential tangency discontinuities imbedded.

**crvTangLines2**( *LineList* )

*Returns*     <curve> Construct a curve from a set of lines ensuring the curve cannot have tangent discontinuities.

*LineList*    <list> List of lines and tangency points described above.

Three curve refinement operators are provided which are curve analogues of the surface refinement operators, so be sure to read the more extensive documentation in the surface section for more information than is given here.

Adding control points to a curve to increase flexibility may be done using **cAddFlex**. The operator attempts to determine a range of the curve corresponding to the *Min* and *Max* range along the specified geometric axis. This means that the curve should extend roughly along the specified axis to get the most accurate results. The *Number* argument specifies (approximately) how many control points will be added to the curve. These control points are spread evenly across the region. Another operator, **cIsolateRegion**, allows a region of the curve to be isolated so that future modifications within the region do not affect the curve outside the region. A point discontinuity can be introduced into a curve using **cFeatureLine** (the name doesn't make much sense in the curve case, but is kept to indicate the surface analogue).

**cAddFlex**( *Crv, Axis, Min, Max, NewPts* )

*Returns*     <curve> Add flexibility to a curve by refining it.  
*Crv*           <curve> Curve to be refined.  
*Axis*          <keyword> Coordinate axis for measurements, 'X', 'Y' or 'Z'.  
*Min, Max*     <number> Lower and upper ends of the region, measured along *Axis*.  
*NewPts*       <number> Number of control points to add.

**cIsolateRegion( Crv, Axis, Min, Max )**

*Returns*     <curve> Refine a curve so that a specified region is isolated from changes to the rest of the curve.  
*Crv*           <curve> Curve to be refined.  
*Axis*          <keyword> Coordinate axis for measurements, 'X', 'Y' or 'Z'.  
*Min, Max*     <number> Lower and upper ends of the region, measured along *Axis*.

**cFeatureLine( Crv, Axis, Value, ContinuityClass )**

*Returns*     <curve> Modify a curve by inserting a discontinuity.  
*Crv*           <curve> The curve to be refined.  
*Axis*          <keyword> Coordinate axis for measurements, 'X', 'Y' or 'Z'.  
*Value*        <number> The coordinate along *Axis* at which the discontinuity is to be introduced.

*ContinuityClass*

<number> How much continuity should be left at the given point.

Curves can be bent just as surfaces can, although the curve should probably lie roughly along the specified axis for the bending to be intuitive. The normal bending operation (see section 8.10.1 [Bending], page 122) can be used for curves, simply by giving a curve as the first argument rather than a surface. The **stretch**, **taper**, and **twist** operations can also be applied to curves by just specifying a curve instead of a surface for the procedures described in the section on operations derived from linear transformations (see section 8.10.2 [Derived From Linear Transformations], page 123).

A variable offset allows an offset curve to be constructed, where the offset distance is determined by a constant, a table, or a user-provided function. The distance information is one of the three forms described for the distance information in **vOffset** for surfaces above.

**cvOffset( Crv, DistInfo )**

*Returns*     <curve> Construct a curve offset from an axis  
*Crv*           <curve | list of curve> Curve to be offset (or list of curves).  
*DistInfo*     <number | table | function> Information describing offset distances (see **vOffset** for more information).

Curve lifting is like **cvOffset**, except that all the points are displaced in a constant direction specified by the user rather than determined by an associated normal vector of the curve.

**cLift( Crv, Direction, DistInfo )**

*Returns*     <curve> Construct a curve by displacing the points of a reference curve in a specific direction.  
*Crv*           <curve | list of curve> Curve to be lifted (or list of curves).  
*Direction*   <geomVector> Direction of lifting.  
*DistInfo*     <number | table | function> Distance of the offsets (see **vOffset** for more information).

A much more accurate offset routine than `cvOffset` is called `qOffset`, and will be described here in detail.

An offset of a curve is produced by offsetting each point of the curve a given distance in a direction that is constant relative to a local frame (coordinate system) on the curve. Typically we think of an offset curve as being defined by:

$$O(t) = C(t) + dN(t)$$

where  $N(t)$  is the principal normal function for  $C(t)$ . The `qOffset` is just a special case of this general definition. We offset by the distance  $d$  in the direction  $(1,0,0)$  relative to the the frenet frame  $\{N(t), B(t), T(t)\}$ , composed of the principal normal, the binormal and the unit tangent.

Using the frenet frame as a local coordinate system causes a number of problems. Chief among them is the fact that the frenet frame may tend to twist about the curve's tangent excessively. Any curve with areas of high torsion will exhibit this behavior; an inflection point being an extreme example. The `qOffset` routine uses local coordinate systems that minimize this twisting effect. Offsetting curves with inflection points presents no problem and offsets of non-planar curves yield intuitive results.

The curve to be offset may be any arbitrary B-spline curve. The resulting offset curve will inherit the parametric information from the curve being offset. The accuracy of the resulting offset curve depends a great deal on the nature of the curve being offset. It is very easy to create self intersecting offset curves. Currently no attempt is made to detect or correct such self intersections.

An offset of a curve composed entirely of straight lines and arcs (that are unit tangent continuous) will yield an exact result. Several offsets of such a curve are shown in Figure 8-31. All other cases yield an approximate result, but the approximation may be made arbitrarily good with refinement of the curve to be offset. Figure 8-32 shows a series of offsets of a more general curve. Figure 8-33 shows the same series of offsets on a refined version of the curve.

Sections of the curve where the curvature is changing rapidly require the most refinement to make the offset curve converge to the correct result. Currently this refinement must be done by hand (eventually the offset routine will automatically refine the curve for you).

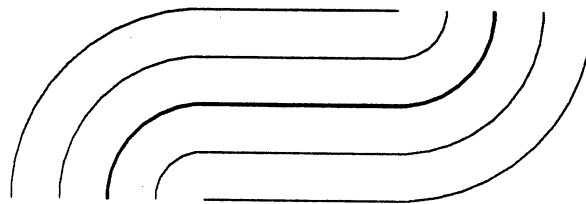
The offset of a curve that is tangent discontinuous poses a fairly difficult problem. Currently a rather simplistic miter joint approach is taken, as shown in Figure 8-34. This approach yields a reasonable result when the tangent discontinuity occurs at the join of two linear (or almost linear) segments.

`qOffset( Crv, RefVec )`

**Returns** <curve> Compute an offset curve of the curve specified.

**Crv** <curve> The curve to be offset.

**RefVec** <number | geomVector> Specifies the offset distance and direction. If `RefVec` is a number, then the first point of the curve is offset a distance  $d$ , in the direction of the curve normal at this point, to yield the first point of the offset curve. Note that  $d$  is a signed distance, i.e., if it is negative the offset will be in the direction opposite the curve normal at the first point. If the curve normal is not defined at the first point of the curve, then the first point along the curve where the normal is defined is located, and the Normal there is "mapped" back to the first point on the curve. If the curve normal is not defined anywhere on the curve, then an arbitrary vector in the normal plane of the first point of the curve is used. If `RefVec` is a vector, then it will be taken directly as the vector by which the first point of the curve is to be offset.

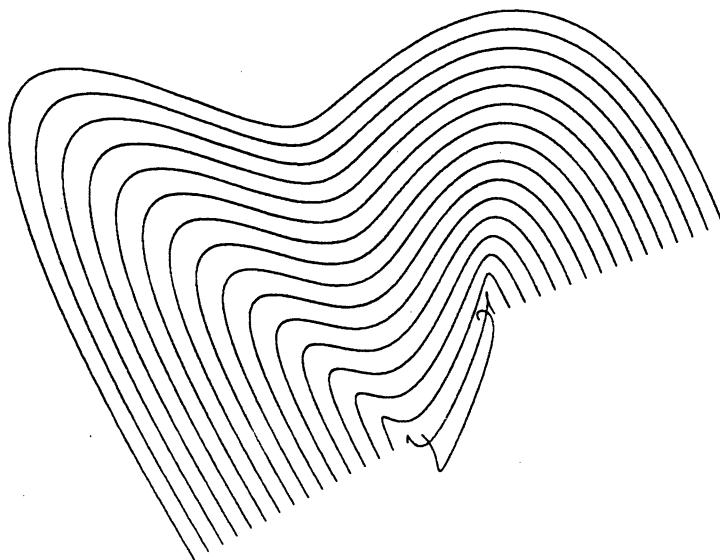


**Figure 8-31: Exact Curve Offsets**

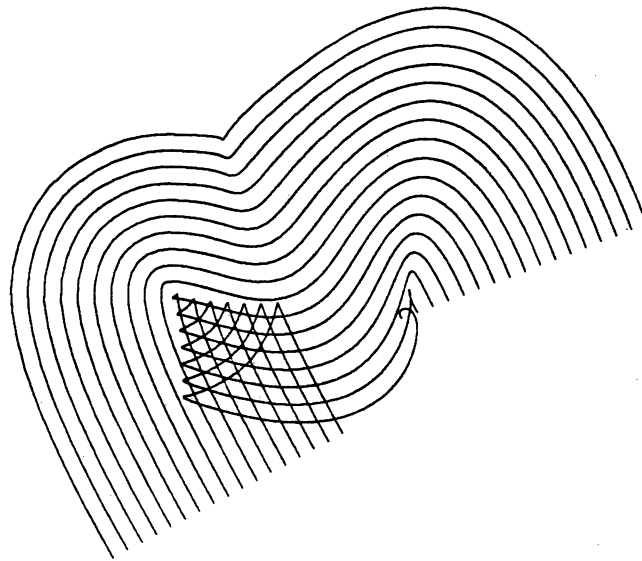
As a general guideline, the most intuitive offsets result from specifying an offset direction that is in the normal plane of the first point of the curve to be offset. At times it may be desirable to specify an offset direction with a component in the direction of the tangent at the first point of the curve. One should be aware however that results may sometimes be a bit unexpected. The resulting offset curve may **not** be unit tangent continuous even though the curve being offset is.

A summary of useful things to keep in mind when using the offset routine includes:

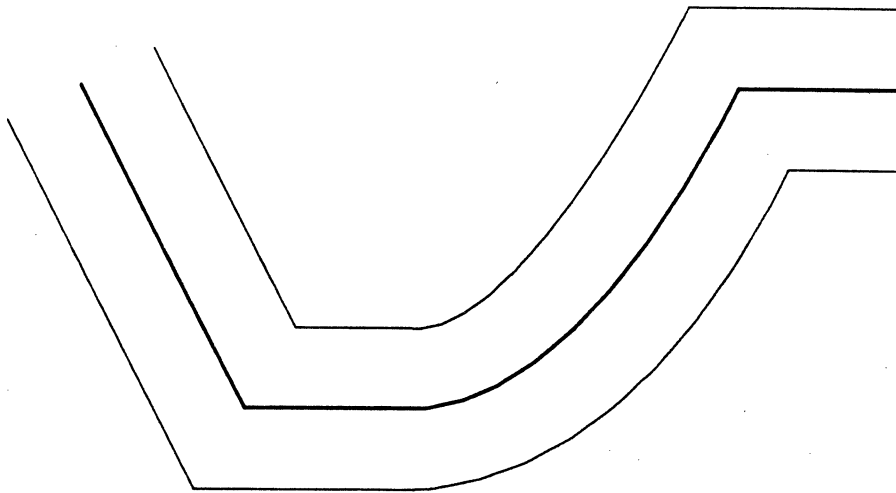
- The curve to be offset may be any arbitrary B-spline curve.
- The resulting offset curve will inherit the parametric information from the curve being offset.
- It is necessary to refine the curve being offset by hand in areas of rapidly changing curvature. It is not necessary to refine the curve on sections that are straight lines or arcs (where the curvature is constant).



**Figure 8-32:** Approximate Curve Offsets Without Refinement



**Figure 8-33: Approximate Curve Offsets With Refinement**



**Figure 8-34: Tangent Discontinuities in Offset**

## 9. Transforming and Grouping Objects

Up to this point, we have described the construction of objects such as points, lines, arcs, curves and surfaces, but have not discussed how to position them in space somewhere other than where they were defined. This section will explain transforming objects, as well as instancing objects and grouping objects for convenience.

### 9.1 Specifying Transformations

Objects may be positioned in space by transforming them using the usual graphics transformations that are representable by 4x4 matrices. These include rotations, scales, and translations as basic building blocks for more complex transformations.

The routines described below construct *matrix descriptor* objects which contain enough information to construct a 4x4 matrix when it is needed for actually transforming some object. Matrix descriptors are used to build up transforms, which are then used to create instances or to transform objects. Most of the time, rather than explicitly creating a transform, you will simply provide a set of matrix descriptors as arguments to the appropriate instancing or transformation operation.

The rotate functions describe rotation about the given axis by an angle specified in degrees. Shorthand names for these routines are **rx**, **ry**, and **rz**.

**rotateX( Angle )**

**Returns** <matDescr> Create a matrix to rotate about the X axis.

**Angle** <number> The angle in degrees to rotate, positive values rotate counter-clockwise.

**rotateY( Angle )**

**Returns** <matDescr> Create a matrix to rotate about the Y axis.

**Angle** <number> The angle in degrees to rotate, positive values rotate counter-clockwise.

**rotateZ( Angle )**

**Returns** <matDescr> Create a matrix to rotate about the Z axis.

**Angle** <number> The angle in degrees to rotate, positive values rotate counter-clockwise.

**rotateLine( Line, Angle )**

**Returns** <matDescr> Create a matrix to rotate about the given line.

**Line** <line> The line about which to rotate (the axis of rotation).

**Angle** <number> The angle in degrees to rotate, positive values rotate counter-clockwise.

The translation functions describe a translation along an axis by the given distance. The **translateXYZ** parameters may either be given as three separate arguments, or as a point or vector. Shorthand names for the latter three are **tx**, **ty**, and **tz**.

**translateXYZ( OffsetX, OffsetY, OffsetZ )**

**Returns** <matDescr> Create a matrix to translate by the given offsets.

*OffsetX, OffsetY, OffsetZ*

*<number>* The offsets in the X, Y, and Z directions, respectively.

**translateXYZ( Offset )**

*Returns* *<matDescr>* Create a matrix to translate by the given offset.

*Offset* *<point | geomVector>* The offset given either as a point or vector.

**translateX( OffsetX )**

*Returns* *<matDescr>* Create a matrix to translate in the X direction.

*OffsetX* *<number>* The distance to translate.

**translateY( OffsetY )**

*Returns* *<matDescr>* Create a matrix to translate in the Y direction.

*OffsetY* *<number>* The distance to translate.

**translateZ( OffsetZ )**

*Returns* *<matDescr>* Create a matrix to translate in the Z direction.

*OffsetZ* *<number>* The distance to translate.

The scaling functions, describe scaling of an axis by the given amount. The **scaleUniform** function applies the same scaling value to all three axes. These names can be abbreviated as **sg**, **s3** or **sxyz**, **sx**, **sy**, and **sz**.

**scaleUniform( ScaleFactor )**

*Returns* *<matDescr>* Create a matrix to uniformly scale by the given factor.

*ScaleFactor*

*<number>* The scaling factor to use.

**scaleXYZ( ScaleX, ScaleY, ScaleZ )**

*Returns* *<matDescr>* Create a matrix to scale each axis by the given factors.

*ScaleX, ScaleY, ScaleZ*

*<number>* The scale factor in the X, Y, and Z directions, respectively.

**scaleX( ScaleX )**

*Returns* *<matDescr>* Create a matrix to scale in the X direction.

*ScaleX* *<number>* The X scale factor.

**scaleY( ScaleY )**

*Returns* *<matDescr>* Create a matrix to scale in the Y direction.

*ScaleY* *<number>* The Y scale factor.

**scaleZ( ScaleZ )**

*Returns* *<matDescr>* Create a matrix to scale in the Z direction.

*ScaleZ* *<number>* The Z scale factor.

A set of "rotate axis" functions rotate one of the X, Y, or Z axes into a specified arbitrary vector, or rotate a specified arbitrary vector into one of the X, Y, or Z axes. Note that one extra degree of freedom remains unspecified (a rotation about the resulting axis) and is fixed arbitrarily. If this extra degree of freedom must be constrained, use the "rotate axis with" functions described



below. A corresponding set of “rotate vector to axis” and “rotate vector to axis with” functions are provided for cases where it is easier to specify the inverse rotation.

**rotateXAxis( Vec )**

**Returns** <matDescr> Create a matrix to rotate the X axis into the given vector.

**Vec** <geomVector> The target vector for the X axis.

**rotateYAxis( Vec )**

**Returns** <matDescr> Create a matrix to rotate the Y axis into the given vector.

**Vec** <geomVector> The target vector for the Y axis.

**rotateZAxis( Vec )**

**Returns** <matDescr> Create a matrix to rotate the Z axis into the given vector.

**Vec** <geomVector> The target vector for the Z axis.

**rotateVectorToXAxis( Vec )**

**Returns** <matDescr> Create a matrix to align the given vector with the X axis.

**Vec** <geomVector> The vector to align.

**rotateVectorToYAxis( Vec )**

**Returns** <matDescr> Create a matrix to align the given vector with the Y axis.

**Vec** <geomVector> The vector to align.

**rotateVectorToZAxis( Vec )**

**Returns** <matDescr> Create a matrix to align the given vector with the Z axis.

**Vec** <geomVector> The vector to align.

The “rotate axis with” functions describe similar transformations, but a second vector argument is specified to fix the remaining degree of freedom. The two given vectors describe a plane in which one of the coordinate planes will lie after transformation. For example, the two arguments of **rotateXAxisWithY** specify the resultant XY-plane. The arguments to each of the “rotate vector to axis with” functions describe the plane to be rotated into the given axes. For example, the two arguments of **rotateVectorToXAxisWithY** describe the plane which is to be rotated into the XY-plane. This describes the inverse of **rotateXAxisWithY** with the same arguments.

**rotateXAxisWithY( NewX, NewY )**

**Returns** <matDescr> Create a matrix which rotates the X axis into the given NewX vector and the Y axis into the plane defined by NewX NewY.

**NewX** <geomVector> The new alignment for the X axis.

**NewY** <geomVector> The new Y axis will be plane defined by this and NewX.

**rotateXAxisWithZ( NewX, NewZ )**

**Returns** <matDescr> Create a matrix which rotates the X axis into the given NewX vector and the Z axis into the plane defined by NewX NewZ.

**NewX** <geomVector> The new alignment for the X axis.

**NewZ** <geomVector> The new Z axis will be plane defined by this and NewX.

**rotateYAxisWithX( NewY, NewX )**

**Returns** <matDescr> Create a matrix which rotates the Y axis into the given NewY

vector and the X axis into the planed defined by *NewY NewX*.  
*NewY* <geomVector> The new alignment for the Y axis.  
*NewX* <geomVector> The new X axis will be plane defined by this and *NewY*.

**rotateYAxisWithZ( *NewY*, *NewZ* )**

**Returns** <matDescr> Create a matrix which rotates the Y axis into the given *NewY* vector and the Z axis into the planed defined by *NewY NewZ*.

*NewY* <geomVector> The new alignment for the Y axis.

*NewZ* <geomVector> The new Z axis will be plane defined by this and *NewY*.

**rotateZAxisWithX( *NewZ*, *NewX* )**

**Returns** <matDescr> Create a matrix which rotates the Z axis into the given *NewZ* vector and the X axis into the planed defined by *NewZ NewX*.

*NewZ* <geomVector> The new alignment for the Z axis.

*NewX* <geomVector> The new X axis will be plane defined by this and *NewZ*.

**rotateZAxisWithY( *NewZ*, *NewY* )**

**Returns** <matDescr> Create a matrix which rotates the Z axis into the given *NewZ* vector and the Y axis into the planed defined by *NewZ NewY*.

*NewZ* <geomVector> The new alignment for the Z axis.

*NewY* <geomVector> The new Y axis will be plane defined by this and *NewZ*.

**rotateVectorToXAxisWithY( *VecToX*, *VecToY* )**

**Returns** <matDescr> Create a matrix which will rotate the given vector *VecToX* into alignment with the X axis and the vector *VecToY* into the XY-plane.

*VecToX* <geomVector> The vector to be aligned with the axis.

*VecToY* <geomVector> The vector to be embedded in the XY-plane.

**rotateVectorToXAxisWithZ( *VecToX*, *VecToY* )**

**Returns** <matDescr> Create a matrix which will rotate the given vector *VecToX* into alignment with the X axis and the vector *VecToY* into the XZ-plane.

*VecToX* <geomVector> The vector to be aligned with the axis.

*VecToY* <geomVector> The vector to be embedded in the XZ-plane.

**rotateVectorToYAxisWithX( *VecToX*, *VecToY* )**

**Returns** <matDescr> Create a matrix which will rotate the given vector *VecToX* into alignment with the Y axis and the vector *VecToY* into the XY-plane.

*VecToX* <geomVector> The vector to be aligned with the axis.

*VecToY* <geomVector> The vector to be embedded in the XY-plane.

**rotateVectorToYAxisWithZ( *VecToX*, *VecToY* )**

**Returns** <matDescr> Create a matrix which will rotate the given vector *VecToX* into alignment with the Y axis and the vector *VecToY* into the YZ-plane.

*VecToX* <geomVector> The vector to be aligned with the axis.

*VecToY* <geomVector> The vector to be embedded in the YZ-plane.

**rotateVectorToZAxisWithX( *VecToX*, *VecToY* )**

**Returns** <matDescr> Create a matrix which will rotate the given vector *VecToX*

into alignment with the Z axis and the vector **VecToY** into the XZ-plane.

**VecToX** <geomVector> The vector to be aligned with the axis.

**VecToY** <geomVector> The vector to be embedded in the XZ-plane.

**rotateVectorToZAxisWithY( VecToX, VecToY )**

**Returns** <matDescr> Create a matrix which will rotate the given vector **VecToX** into alignment with the Z axis and the vector **VecToY** into the YZ-plane.

**VecToX, VecToY**

<geomVector> The vector to be aligned with the axis.

**VecToY** <geomVector> The vector to be embedded in the YZ-plane.

Transforms are lists of matrix descriptors which describe some desired transformations. The **transform** function which is used to create transforms may have no arguments, in which case the transformation described is just the identity. Or it may have any number of the matrix descriptors described above as arguments. In addition, a point appearing in the list of arguments is accepted as a shorthand form of **translateXYZ**. The resulting transformation is merely the result of applying all the matrix descriptors one after the other.

**transform( Descr1, ... )**

**Returns** <transform> Concatenate a sequence of matDescr's into a single transform. If no matDescr's are given return the identity transform.

**Descr1, ...** <opt point | matDescr> A (possibly empty) sequence of matrix descriptors or points. A point is treated as a short-hand for **translateXYZ**.

Transforms may also be used to build "transformation trees" which have more structure than a linear list. A transform is also a matrix descriptor, and can be used anywhere a matrix descriptor is required.

If you should really want to look at the 4x4 matrix which represents the transformation you have described, use **extractMatrix**.

**extractMatrix( Transform )**

**Returns** <matrix4x4> Extrac the 4x4 matrix defined by the given transform.

**Transform** <maDescr | transform> The matrix descriptor or transform to examine.

### Editing Transforms

Once a transform is created, the list of matrix descriptors it contains may be modified using functions which add or delete matrix descriptors in the transform. These functions may also be used to edit the transforms associated with instanced objects, which are described later in this chapter (see section 9.3 [Instancing Objects], page 149). Note that these functions modify the given transform, but do not return the modified transform as their return value.

The **appendDescriptor** routine adds a new descriptor to the end of the list of descriptors, while **insertDescriptor** inserts the new matrix descriptor after the Nth one in the list. The first matrix descriptor in the list is numbered 1. So if you wish to insert a new descriptor at the beginning of the list of descriptors, use  $N = 0$ . If  $N$  is greater than the number of descriptors in the descriptor list, then the descriptor specified is appended to the end of the descriptor list.

**appendDescriptor( OrigDescr, NewDescr )**

**Returns** <transform> Append a new matrix descriptor to a descriptor set.

**OrigDescr** <transform | instance> The original descriptor set.

**NewDescr** <matDescr> The new descriptor to append.

**insertDescriptor**( *OrigDescr*, *N*, *NewDescr* )

*Returns*     <transform> Insert a new matrix descriptor into a descriptor set.  
*OrigDescr*   <transform | instance> The original descriptor set.  
*N*             <number> The descriptor index which the new descriptor should follow.  
*NewDescr*   <matDescr> The new descriptor to add.

To remove a matrix descriptor from a transform, use **deleteDescriptor** to delete the *N*th matrix descriptor in the list. If *N* is greater than the number of descriptors in the descriptor list or *N* = 0, then the transform is not changed. To replace an existing matrix descriptor in the transform with a new one, use **replaceDescriptor**. If *N* is greater than the number of descriptors in the descriptor list or *N* = 0, then the transform is not changed. The function **indexDescriptor** returns the *N*th matrix descriptor in the transform. The first matrix descriptor in the list is counted as *N* = 1. If *N* is greater than the number of descriptors in the descriptor list or *N* = 0, then Nil is returned.

**deleteDescriptor**( *OrigDescr*, *N* )

*Returns*     <transform> Remove a matrix descriptor from a descriptor set.  
*OrigDescr*   <transform | instance> The descriptor set to delete from.  
*N*             <number> The descriptor number to remove, the first descriptor has index 1.

**replaceDescriptor**( *OrigDescr*, *N*, *NewDescr* )

*Returns*     <transform> Replace one descriptor with another in a descriptor set.  
*OrigDescr*   <transform | instance> The descriptor set to modify.  
*N*             <number> The descriptor index to replace, the first descriptor has index 1.  
*NewDescr*   <matDescr> The new descriptor to insert.

**indexDescriptor**( *OrigDescr*, *N* )

*Returns*     <matDescr> Extract the *N*th descriptor from the descriptor set.  
*OrigDescr*   <transform | instance> The descriptor set to examine.  
*N*             <number> The index of the descriptor to return, the first descriptor has index 1.

## 9.2 Transforming Objects

All the transformations described in the previous section are not useful unless we can apply those transformations to objects which we construct in **shape\_edit**. There are two ways of doing this. One is to apply the transformation by multiplying all the points in the object by the 4x4 matrix which represents the transformation and represent the resulting object using those transformed points. The other is to create an "instance" of the object, associating the transformation with the object. The actual transforming of the points will be done when needed by **shape\_edit**. There are many advantages to using instances, and you should use instances to transform objects unless you have a good reason to want to force the points to be transformed immediately. An important advantage to instances is that you will often want to adjust the transformations after examining the results on a display. Since many display devices have special purpose hardware for transforming points, your response time will often be significantly faster if **shape\_edit** only has to send down a new matrix and let the hardware transform all the data. Not only is the special purpose hardware likely to be much faster than the host; the amount of data which has to be transmitted is generally much less. However, not all the operations in the system that operate on objects you can instance,

operate on instances of those objects. So occasionally you must just push the points through the transformation matrix.

Should you need to transform an object without creating an instance, the function **objTransform** will perform the transformation. The matrix descriptors can be any of those which were described in the previous section. There is no need to actually construct a transform as this is done implicitly by **objTransform**. On rare occasions, it is easier to specify the inverse of the desired transformation. Use **invObjTransform** to have the object transformed by the inverse of the specified list of transforms.

**objTransform**( *Obj*, *Descr1*, ... )

*Returns*     <object> Transform an object given a set of transformations.

*Obj*             <object> The object to transform.

*Descr1*, ... <matDescr> A sequence of matrix descriptors to apply to the object.

**invObjTransform**( *Obj*, *Descr1*, ... )

*Returns*     <object> Transform an object through the inverse of a set of transforms.

*Obj*             <object> The object to transform.

*Descr1*, ... <matDescr> A sequence of matrix descriptors which are concatenated, then their inverse is taken and applied to the object.

### 9.3 Instancing Objects

An instance of an object is created by associating the object with a transform using **instance**. If no matrix descriptors are provided, the instance has an identity transformation associated with the object. As for the transform construction, a matrix descriptor may be a point which is a short form for the **translateXYZ** descriptor. There is no need to construct a transform (although transforms may be used as well); it is implicit in the instance constructor when a set of matrix descriptors are provided.

**instance**( *Obj*, *Descr1*, ... )

*Returns*     <instance> Generate an instance of an object given a sequence of matrix descriptors.

*Obj*             <object> The object to instance.

*Descr1*, ... <matDescr> A sequence of matrix descriptors to apply to the object.

**instance**( *Obj*, *Transform* )

*Returns*     <instance> Generate an instance of an object given a transform.

*Obj*             <object> The object to instance.

*Transform* <transform> The transform to apply to the object.

The **extractMatrix** routine described above, as well as all the transformation editing functions (see section 9.1 [Specifying Transformations], page 143) may be used for instances as well as transforms. These include **extractMatrix**, **appendDescriptor**, **insertDescriptor**, **deleteDescriptor**, **replaceDescriptor**, and **indexDescriptor**. In addition, **replaceObject** replaces the instanced object in the instance object with the specified object specified. It associates the same transformation with a different object.

**replaceObject**( *Instance*, *Obj* )

*Returns*     <instance> Instance of given object with same transformation as original instance.  
*Instance*     <instance> The original instance (object part will be replaced, transformation part kept).  
*Obj*           <object> New object to put in the instance.

## 9.4 Grouping Objects

Groups are often useful for manipulating sets of objects as a unit. For example, if a wheel object has been instantiated to form the four wheels of a vehicle, it may be useful to control display of the four wheels as a group rather than individually.

The **group** constructor forms the given objects into a group. If you already have some or all of the objects of the group collected in a list or vector, you may just give the list or vector as an argument, and the **group** constructor will add all those objects to the group as well.

**group**( *Obj1*, ... )

*Returns*     <group> Collect a set of objects together and treat them as a "single" object.  
*Obj1*, ...     <object | listOf object | vectorOf object> A sequence, list, or vector of objects to be collected into a group.

An object may be added to the group using **addToGroup**, or deleted from the group with **deleteFromGroup**. If a given object is a member of the group, **memberOfGroup** will return a true value.

**addToGroup**( *Group*, *Obj* )

*Returns*     <group> Add an object to an existing group.  
*Group*       <group> The group to add to.  
*Obj*          <object> An object to add to the group.

**deleteFromGroup**( *Group*, *Obj* )

*Returns*     <group> Remove an object from an existing group.  
*Group*       <group> The group to modify.  
*Obj*          <object> The object to remove.

**memberOfGroup**( *Group*, *Obj* )

*Returns*     <boolean> Determine if the given object is in the group.  
*Group*       <group> The group to examine.  
*Obj*          <object> The object to look for.

## 9.5 Output Optimization for Groups & Instances

If many instances of a single object are made, the output from **shape\_edit** can become quite voluminous. This section describes a mechanism which allows occurrences of particular objects to be referenced by name in Alpha\_1 formatted text files when embedded in groups and instances. The geometric data of the object need be printed only once, instead of expanded in line for each reference. This feature is also useful for debugging complex structures of groups and instances. A simpler geometric object can be substituted for a complex one until the transformations are completely worked out.

An object which is going to be instanced or included in a group may be tagged as a "library entry" before the instance or group is created, using the `libraryEntry` function.

`libraryEntry( Obj )`

*Returns*     <Nil> Tag an object as a library entry.

*Obj*            <object> The object to tag.

When instances and groups which contain that object are dumped using `dumpA1File` (which will be described in a later chapter), only the name of the object is dumped — not its geometric description. When using that file in an input stream to another program, the actual object must precede any references to it.

As an example, suppose we have a complex object called `Part`. Then

```
libraryEntry( Part );           % Mark the part as a libraryEntry.
```

```
A := instance( Part, tx 10 );   % Create instances of it.
```

```
B := instance( Part, tx 20 );
```

```
dumpA1File( Part, "part.a1" );
```

```
dumpA1File( list( A, B ), "double.a1" );
```

We could also have said

```
dumpA1File( list( Part, A, B ), "double.a1" );
```

which would have the same effect, except that a simplified version of the part can't be substituted into the object stream in place of the complex one.





## 10. Defining New Object Types

Parametric types are a quick means of extending the set of available model object types to include new, higher-level objects whose geometry can be expressed in terms of model object types already known to the modeling system. Parametric types may be defined at run-time as “interpreted” definitions in `shape_edit`, or may be saved in a file, compiled and rapidly loaded. In either case, the parametric type is understood as part of the family of object types which make up `shape_edit` models.

Each object type has a set of named “slots” which store the “parameters” that make each object of the type different from others. Parameters may be anything the modeling system understands: dimensions, points, vectors, curves, surfaces, shells, objects which are members of parametric types, and so on. The parameters are stored in an object when it is created, and become the arguments to the `makeGeom` procedure described below, which will generate the actual geometry of the object. Thus, the only restriction on the parameters of a particular parametric object type is that the parameter values can be handled properly by the `makeGeom` routine.

Parametric types are defined with `defParamType`. The first argument is the name of the parametric type, and the rest of the arguments are the parameter names.

```
defParamType( TypeName, Param1, ... )
```

Returns <Nil> Defines a new parametric type.

TypeName

<symbol> The name of the new parametric type.

Param1, ...

<symbol> The name for each field of the parametric type. There may be a parameter options associated with each field as described below.

For example:

```
defParamType( Sphere, Ctr, Rad );
```

Following each field name there may be a single numeric parametric specifier for that parameter. This provision is made because numeric parameters may need a little more specification of how the parameter is to be handled when the object is transformed. Angles and “pure numbers” (e.g., the number of teeth on a gear) will be left alone. Positions and sizes need to be specified so they can be scaled and translated properly.

Numeric parameter types are:

**Size** Perform scaling, but not translation.

**XPosition** A location along the X Axis.

**YPosition** A location along the Y Axis.

**ZPosition** A location along the Z Axis.

An example of the use of a numeric parameter declaration would be for the radius of a sphere:

```
defParamType( Sphere, Ctr, Rad( Size ) );
```

Or, to specify a parametric type for a box with faces parallel to the coordinate planes you could say

```
defParamType( coordinateBox, xMin( XPosition ), xMax( XPosition ),
              yMin( YPosition ), yMax( YPosition ),
              zMin( ZPosition ), zMax( ZPosition ) );
```

A parametric type may have an implied position. For example, it may always construct its geometry in the plane  $z=0$ . Since this positioning is not controlled by the value of any slot, but rather in the semantics of the **makeGeom** method, this must be declared to cause **mapObj** to behave properly. The declarations **ImplicitXPosition**, **ImplicitYPosition**, and **ImplicitZPosition**, are available as options to the first argument of the **defParamType** macro. For example, a parametric type to represent a circle in the  $z=0$  plane might have slots for the center and radius. It would have a declaration of the form:

```
defParamType( Circ( ImplicitZPosition ), Ctr, Radius( Size ) );
```

A constructor function is defined by **defParamType** which has the same name as the *TypeName*, and which takes the parameters as arguments. So a sphere object is created by:

```
sphere( Origin, .5 );
```

(Note for more advanced users: a **!InheritFrom** option of the type name overrides the default inheritance, which is to inherit from the **paramObj** type, which itself inherits from **modelObj**.)

### MakeGeom Method Procedures

When any new model object type is defined, many *generic methods* of the object type are defined which tie it into the modeling system. A method procedure is a special kind of procedure for objects. A *method* is an operation that has a uniform meaning for many different kinds of objects. For example, the **reverseObj** operation is actually a method which has "reverse the orientation of the object" as its general meaning. The actual implementation of reversing orientation for any particular object type is contained in its method procedure. Thus, there are many **reverseObj** method procedures, and when **reverseObj** is called on a particular object, the system figures out which actual method procedure to call.

These details are taken care of automatically for parametric object types once a **makeGeom** method procedure is defined. This procedure will be called to construct the "geometry" of an object, given the stored parameters, when the geometric information is needed. For example, **reverseObj** may not know anything at all about a particular parametric type object which you have defined, but if you provide the **makeGeom** method, **reverseObj** can call that and then apply the known **reverseObj** procedures to the geometry which was constructed.

The **makeGeom** method procedure can use any of the constructors, extractors, and Lisp functions available in the modeling system in constructing a lower-level representation of the object. In fact, a good way of building parametric object types is to build a "prototype" of the object type interactively, starting by assigning values to variables which will become the names of the parameters of the object. Then execute a **defParamType** and wrap a **makeGeom** method procedure around the construction to associate it with the type. The **makeGeom** method procedure usually returns a shell object. Although any object type is legal, only shells will be able to be used later for boolean combination operations.

The geometry is constructed by calling the **makeGeom** procedure when it is first needed by a method (for example, to display the object). The geometry is then stored until one or more of the parameters is changed so it is not reconstructed each time it is referenced.

For example:

```
internalGeometryMethod procedure Sphere makeGeom();
begin
  % Just instance a unit sphere appropriately.
  instance( UnitSphere, scaleUniform Rad, translateXYZ Ctr )
end;
```

Note the name of the object type in the first line of the definition. Also, the **internalGeometryMethod** qualifier to the **procedure** keyword is important. Note also that the parameters of the object are referenced as variables in the construction expression, since it will be invoked as a method of the object type.

You may want to define an optional **init** method of the object type, which will be called during the construction of a new object, **after** the object is initialized by filling in the parameters provided in the call. A good thing to do in an **init** method is to check the types and values of parameters, invoking the **stdError** function with a message string explaining any errors found. For example:

```
internalGeometryMethod procedure Sphere init( ArgList );
begin
  if Center ne '!*Unbound!* and not pointP Center then
    stdError bldMsg( "Sphere: Center must be a point.%n" );

  if Radius ne '!*Unbound!* and not numberP Radius then
    stdError bldMsg( "Sphere: Radius must be a number.%n" );
end;
```



## 11. Applications

A few preliminary application interfaces have been built within **shape\_edit**. This chapter describes an animation package, a package for adding dimensions to the display, a package for creating manufacturing features which commonly occur in mechanical parts, a package for producing numerical control machining instructions, and an interface package for finite element analysis.

### 11.1 Animation Utilities

IOU - Animation Utilities (gm)

### 11.2 Adding Graphical Dimensions

Some basic features for dimensioning the objects modeled in Alpha\_1 are provided to make output from Alpha\_1 have clearer meaning. This is a preliminary package which may be used eventually to provide a more serious engineering drawing package. No attempt has been made to create dimensioning and tolerancing features according to the developing ANSI standards. Also, Alpha\_1 currently doesn't support creation of geometric objects where text size is taken into account.

These dimensioning features are intended to be easy to use for users who do not want to spend a lot of time customizing a drawing. Dimensions are available in a variety of forms, so a drawing can be clear and uncluttered, regardless of the geometry being represented. If the structure of dimensions is inflexible, a drawing will be impossible or difficult to read. Currently, creating dimensions is only supported in the programmer's (command driven) interface, and not in the graphical user interface.

One goal of this package was that the dimensioning features should be well-adapted to hard copy printing (e.g., Postscript files). However, the displays supported under the **shape\_edit** interactive environment allow the user to dynamically change views of three dimensional objects. In this interactive context, it would be taking something away from the resources offered by the system to simply project all dimension features onto a single plane. As a result, most of the dimension features are implemented as three dimensional wire-frame objects. This raised some new issues, because traditional media for displaying dimensions are two dimensional.

It is worth noting that the different measures can be applied in different contexts. For example, the measurement of a radius is usually considered in an absolute context. Angles can be considered in an absolute sense, or they can be measured according to their projections onto planes. Linear dimensions can be useful in a three dimensional sense, or according to their projections onto planes or lines.

There are a few things to watch out for. If your dimension objects look odd, make sure you know whether you are viewing a three dimensional dimension in perspective or not. Note: this dimensioning package was written before arcs and circles were extended to 3D, and so dimensions on 3D arcs and circles may be incomplete.

#### 11.2.1 Attributes for Dimensions

There are three types of dimension objects provided: angles, linear distances, and radii (or diameters). For each of the dimension object types, there is a set of attributes which can be associated with an instance of the type. These attributes affect the geometric layout of the dimension objects.

For example, a radius dimension can be drawn inside or outside the arc being measured. This flexibility is necessary for making good drawings.

To make the attributes easier to specify, there is a global set of default attributes for each of the dimension types. These default attributes are inherited by specific dimension objects. (These global objects are named **AngDimDefaultAttrs**, **DiaDimDefaultAttrs**, and **LinDimDefaultAttrs**.) The attributes of a specific dimension object, when present, will always override the default values. The default attributes for a type of dimension object can be changed, so the environment for creating dimension objects can be customized. The global default attributes can be brought down to a single dimension object through a command to adopt the global default values.

There are a few important ideas related to the attribute inheritance scheme used. The use of the default attributes helps keep the number of objects down. Not all attributes are applied all the time — specification of certain attributes implies that others may be ignored or overridden. For example, you cannot have the message of a dimension outside the dimension if the arrows are inside the dimension. If a user tries to set an invalid dimension attribute, then an error message is printed to the terminal.

### Attributes of Dimensions

The attributes for the various dimension types and their meanings are described below. The type of each attribute is given. Note that some attributes are not meaningful for all three dimension types. In these cases, the words “angular”, “radius”, and “linear” in parentheses indicate for which of the types the attribute has meaning.

#### 'ArrowAngle

<number> (angular) The angle (degrees) of the arc for the arrows of the angular dimension. Only applies when the attribute

#### 'ArrowLength

<number> (radius, linear) For radius dimensions, the value to scale the radius by to calculate the distance out from the center where the dimension should be placed. Only applies when the attribute **ArrowsOutside!?** is true. For linear dimensions, it is the absolute distance that the dimension should be away from the points, perpendicular to the direction being measured.

#### 'ArrowsOutside!?

<boolean> (angular,linear) Whether the layout of the dimension should leave the arrows outside.

#### 'Diametric!?

<boolean> (radius) Whether the dimension should measure the diameter, as opposed to the radius.

#### 'DimOffsetDir

<vector> (linear) Because the linear dimensions exist in three dimensions, there exists an entire plane perpendicular to the direction being measured. This vector allows the user to specify in what direction the dimension should be offset.

#### 'DimOffsetDistance

<number> (linear) The distance from the points being measured that the dimension should be placed.

#### 'DimOutside!?

<boolean> (radius) Whether the layout of the dimension should leave the arrows and text on the outside.

- 'DirOfMeasure  
     <geomvec> (linear) The direction the measurement should be taken along.
- 'MsgExtraArgsList  
     <list> List of extra arguments to be sent to the **bldmsg** function to be part of the dimension message. For angular dimensions, the value of the angle in degrees is always the first argument to **bldmsg** for the message generation. For radius dimensions, the value of the radius or diameter is the first argument. For linear dimensions, the value of the distance along the direction of measurement is the first argument.
- 'MsgFormatString  
     <string> Format string for the dimension message.
- 'MsgHorizontalOffset  
     <number> (radius, linear) Horizontal distance that the message should be offset.
- 'ParameterValue  
     <number> (radius) This determines the location on the arc or circle where the dimension is placed.
- 'Radius  
     <number> (angular) The radius of the angular dimension. It describes how far from the centerpoint the arrows should be.
- 'SpaceSize  
     <number:0.0-1.0> When the attribute **ArrowsOutside!?** is Nil, this determines the size of the white space around the message.
- 'SpaceLocation  
     <number:0.0-1.0> When the attribute **TextOutside!?** is Nil (or **DimOutside!?** for radius dimensions), determines where the message is placed.
- 'TextOutside!?  
     <boolean> (angular, linear) Determines if the dimension text is inside or outside the measured region. For linear dimensions, if the arrows are on theinside, then the text is forced to be on the inside as well.

## 11.2.2 Creating Dimensions

### Angular Dimensions

To create dimension objects, use **angDimAttr**, **diaDimAttr**, and **linDimAttr**. Angular dimensions require a point and two direction vectors. Diameter or radius dimensions require an arc or circle, while linear dimensions require two points. Attributes in name/value pairs may follow the required arguments for each.

**angDimAttr**( *Point*, *Direction0*, *Direction1*, *AttrName*, *AttrVal*, ... )

*Returns*     <angDim> An angular dimension object.

*Point*        <euclidPoint> The center point for the dimension.

*Direction0*, *Direction1*

              <geomVector> Directions for the angle.

*AttrName*     <keyword> One of the keywords described above.

*AttrVal*       <any> The value for the previous keyword.

**diaDimAttr**( *ArcOrCircle*, *AttrName*, *AttrVal* )

*Returns*     <diaDim> A diameter or radius dimension object.

**ArcOrCircle**

<arc | circle> The object which this dimension describes.

**AttrName** <keyword> One of the keywords described above.

**AttrVal** <any> The value for the previous keyword.

**linDimAttr**( *Point0*, *Point1*, *AttrName*, *AttrVal* )

**Returns** <linDim> A linear dimension object.

*Point0*, *Point1*

<euclidPoint> The end points which the dimension is measuring between.

**AttrName** <keyword> One of the keywords described above.

**AttrVal** <any> The value for the previous keyword.

To change one of the default values for angular dimension objects, use the **changeDimDefaultAttr** functions. This places the name-value pair on the global dimension attribute list. For example, you may decide that you would like certain characteristics to be the defaults when angular dimensions are created. You might place a block like this at the top of your **shape\_edit** file:

```
{
  changeDimDefaultAttr( 'Ang', 'ArrowsOutside', T );
  changeDimDefaultAttr( 'Ang', 'TextOutside', T );
  changeDimDefaultAttr( 'Ang', 'MsgFormatString', "%w DEGREES" );
};
```

The result of this block would be to cause those attribute values to be used when angular dimension objects are created.

**changeDimDefaultAttr**( *DimType*, *AttrName*, *AttrValue* )

**Returns** <Nil> Change one of the default attributes for one of the dimension types.

**DimType** <keyword> One of 'Ang', 'Dia', or 'Lin'.

**AttrName** <keyword> One of the keywords described above.

**AttrVal** <any> The value for the previous keyword.

To change the value of an attribute on a specific angular dimension, use the **changeDimAttr** functions.

**changeDimAttr**( *TheDim*, *AttrName*, *AttrValue* )

**Returns** <Nil> Change one of the dimension attributes of a particular dimension object.

**TheDim** <angDim | linDim | diaDim> The object to be modified.

**AttrName** <keyword> One of the keywords described above.

**AttrVal** <any> The value for the previous keyword.

Finally, the **adoptDimDefaultAttrs** function cause the particular dimension object to take on all the global default attributes where any attributes have not been specified. In other words, if a change is going to be made to the global default attributes, but one would like to keep currently existing dimensions as they are, the global default values should be adopted by the particular dimension object.

**adoptDimDefaultAttrs**( *TheDim* )

**Returns** <Nil> Copy the default attributes for this dimension into its attribute list.

**TheDim** <angDim | linDim | diaDim> The dimension object to be changed.



### 11.2.3 A Dimensioning Example

The following example may serve to illustrate the use of the dimensioning operations. The cross-section of an aluminum soda can is constructed first, and dimensions added. The dimensions are shown in Figure 11-1.

```
% Construction of cross-section of coke can.
{
    SideLine ^= lineVertical( CanRad := 1.4 );
    RimLine ^= lineVertical( CanRad - 0.2 );
    TopLine ^= lineHorizontal( TopEdge := 4.7 );
    TopPoint ^= ptIntersect2Lines( RimLine, TopLine );
    TopSlantLine ^= linePtAngle(
        ptIntersect2Lines( lineHorizontal( TopEdge - 0.25 ),
                           RimLine ),
        90 + 45 );
};

{
    TopRimArc ^= arcRadTan2Lines( TypicalRad := 0.1,
                                   TopSlantLine, RimLine );
    BotRimArc ^= arcRadTan2Lines( TypicalRad, SideLine, TopSlantLine );
    BottomLine ^= lineHorizontal( 0 );
    BottomOuterArc ^= arcRadTan2Lines( 0.5, BottomLine, SideLine );

    BottomSlantLine ^= linePtAngle( ptIntersect2Lines(
        lineVertical( CanRad-0.6 ),
        BottomLine ),
        90 + (90-75) );
    BottomSmallArc ^= arcRadTan2Lines( TypicalRad,
        reverseObj BottomSlantLine,
        BottomLine );

    BottomDepth := 0.4;
    BottomArcP1 ^= ptIntersect2Lines( lineHorizontal BottomDepth,
        BottomSlantLine);
    BottomArc ^=
        arcEndCornerEnd( ptIntersect2Lines(
            lineHorizontal BottomDepth,
            lineVertical( 0.0 ) ),
            ptIntersect2Lines(
                lineHorizontal BottomDepth,
                linePtAngle( BottomArcP1, 90 + 60 )),
            ptIntersect2Lines(
                BottomSlantLine,
                lineHorizontal( 0.2 ) ) );
};

% Construct and display the final curve constructed...
{
    CokeProfile ^= profile( BottomArc, BottomSmallArc, BottomOuterArc,
```

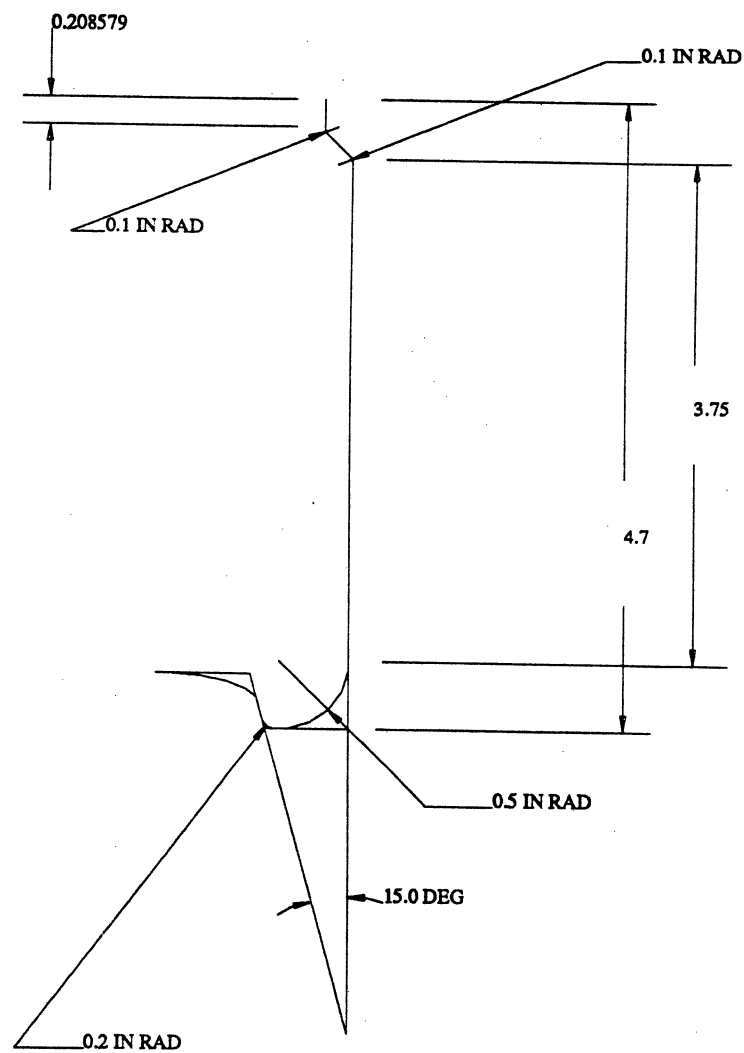


Figure 11-1: Dimensions Examples

```

        BotRimArc, TopRimArc, TopPoint );
};

PtTopOfSideLine ^= ptIntersect2Lines( SideLine, TopSlantLine );
PtBotOfSideLine ^= arcEnd BottomOuterArc;

SideLinDim ^= linDimAttr( e3 PtTopOfSideLine, e3 PtBotOfSideLine,
        'DimOffsetDistance, -2.5 );

BotOutRadDim ^= DiaDimAttr( BottomOuterArc,
        'Diametric!?, Nil,
        'MsgFormatString, "%w IN RAD",
        'DimOutside!?, t,
        'ArrowLength, 1.0);

BotSmallRadDim ^= DiaDimAttr( BottomSmallArc,
        'MsgFormatString, "%w IN RAD",
        'DimOutside!?, t,
        'ArrowLength, 3.0,
        'MsgHorizontalOffset, 0.5);

TopRimRadDim ^= DiaDimAttr( TopRimArc,
        'Diametric!?, Nil,
        'MsgFormatString, "%w IN RAD",
        'ArrowLength, 2.0,
        'DimOutside!?, T,
        'MsgHorizontalOffset, 0.25);

BotRimRadDim ^= DiaDimAttr( BotRimArc,
        'Diametric!?, Nil,
        'MsgFormatString, "%w IN RAD",
        'ArrowLength, 2.0,
        'MsgHorizontalOffset, 0.25,
        'DimOutside!?, t);

PtWayAtBottom ^= ptIntersect2Lines( SideLine, BottomSlantLine );

BotAngDim ^= angDimAttr( PtWayAtBottom, Ydir, dirOfLine(BottomSlantLine),
        'ArrowsOutside!?, t,
        'TextOutside!?, t,
        'MsgFormatString, "%w DEG" );

PtBotOfBotRimArcLine ^= arcStart BottomSmallArc;

ExtraLine ^= polyline( list( e3 PtBotOfSideLine,
        e3 PtWayAtBottom,
        e3 PtBotOfBotRimArcLine ) );

ExtraBottomPt ^= ptIntersect2Lines( BottomLine, SideLine );
ExtraTopPt ^= ptIntersect2Lines( TopLine, SideLine );

```

```

BigLinDim := linDimAttr( e3 ExtraBottomPt, e3 ExtraTopPt,
                        'SpaceLocation, 0.3 );

TopSmallLinDim ^= linDimAttr( e3 TopPoint, e3 arcEnd TopRimArc,
                              'TextOutside!?, t,
                              'ArrowsOutside!?, t );

DimStuff ^= group( SideLinDim, BotOutRadDim,
                   BotSmallRadDim, TopRimRadDim,
                   BotRimRadDim, BotAngDim,
                   CokeProfile, ExtraLine,
                   BigLinDim, TopSmallLinDim );

```

### 11.3 Mechanical Features

Several types of manufacturing features are commonly encountered when designing mechanical parts. This section describes a group of objects built using the parametric type facility which make it easier to construct models which contain those features. The features currently provided include **counterBore**,

**counterBore**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth* )

*Returns*     <counterBore> Constructs a counterBore parametric type.

*CtrPt*       <euclidPoint> The center point.

*OutDia*, *InDia*

              <number> Outer and inner diameter of the counter-bore.

*OvDepth*, *InDepth*

              <number> Overall and inner depths of the counter-bore.

**inverseCounterBore**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth* )

*Returns*     <inverseCounterBore> Constructs an inverseCounterBore parametric type.

*CtrPt*       <euclidPoint> The center point.

*OutDia*, *InDia*

              <number> Outer and inner diameter of the inverse counter-bore.

*OvDepth*, *InDepth*

              <number> Overall and inner depths of the inverse counter-bore.

**counterSink**( *CtrPt*, *SinkDia*, *Angle*, *HoleDia*, *OvDepth* )

*Returns*     <counterSink> Constructs a counterSink parametric type.

*CtrPt*       <euclidPoint> Center point for the counter-sink.

*SinkDia*     <number> Diameter of the counter-sink.

*Angle*       <number> Angle of the counter-sink.

*HoleDia*     <number> Diameter of the hole part of the counter-sink.

*OvDepth*     <number> Overall depth of the counter-sink.

**counterDrill**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth*, *Angle* )

*Returns*     <counterDrill> Constructs a counterDrill parametric type.

*CtrPt*       <euclidPoint> Center point for the counter-drill.

*OutDia, InDia*

<number> Outer and inner diameters for the counter-drill.

*OvDepth, InDepth*

<number> Overall and interior depth for the counter-drill.

*Angle* <number> Angle of the counter-drill.

**slottedHole( CtrPt, TotalLength, Width, Angle, Depth)**

*Returns* <slottedHole> Constructs a slottedHole parametric type.

*CtrPt* <euclidPoint> Center point of the hole.

*TotalLength*

<number> Length of the slotted hole.

*Width* <number> Maximum width of the hole.

*Angle* <number> Angle for slanted sides of the hole.

*Depth* <number> Depth of the hole.

**electricConnectorSet( CtrPt, LargeDia, Height, SmallDia, Depth, Space )**

*Returns* <elecConnSet> Constructs an electricConnectorSet parametric type.

*CtrPt* <euclidPoint> Center point of the connector set.

*LargeDia* <number> Diameter of the large center hole.

*Height* <number> Height of the large center hole.

*SmallDia* <number> Diameter of the small holes.

*Depth* <number> Depth of the holes.

*Space* <number> Spacing of the holes.

**rectangularPocket( UpperLeftCorner, TotalLength, Width, Depth, CornerRadius, BottomRadius )**

*Returns* <recPocket> Constructs a rectangularPocket parametric type.

*UpperLeftCorner*

<euclidPoint> Base point for pocket, upper left corner.

*TotalLength, Width, Depth*

<number> Overall dimensions of the pocket.

*CornerRadius*

<number> Radius of the four corner edges.

*BottomRadius*

<number> Radius of the four bottom edges.

**linearPattern( Object, StartPt, NumberOfObjects, Space, Angle)**

*Returns* <linearPattern> Constructs a linearPattern parametric type.

*Object* <object> The object to make the pattern out of.

*StartPt* <euclidPoint> New origin for first object.

*NumberOfObjects*

<integer> Number of objects in the pattern.

*Space* <number> Spacing between the objects.

*Angle* <number> Angle between the objects.

**radialPattern( Object, CtrPt, NumberOfObjects, StartAngle, EndAngle, PitchDia )**

*Returns* <radialPattern> Constructs a radialPattern parametric type.

*Object* <object> The object to make the pattern out of.

*CtrPt*        <euclidPoint> The center of the radial pattern.  
*NumberOfObjects*        <integer> Number of objects in the pattern.  
*StartAngle, End Angle*        <number> Beginning and ending angle for the pattern.  
*PitchDia*        <number> Diameter of the radial pattern.

## 11.4 NC Machining

Some support for generating numerical control machining "tapes" is available in `shape_edit`. To use the functions described in this section, it is necessary to first execute

```
load nc;
```

The NC package is based on a two stage process: commands are created for an abstract milling machine, and are then translated for the desired actual machine. The only output machine currently supported is a Monarch Cortland VMC45 milling machine operating in a 3-axis mode. Many commands can (currently) only be issued by specifying their "G-codes" in an "escape" block, so that the desired machine independence is still somewhat lacking.

Each NC command block contains a state vector, completely specifying the desired abstract machine state for execution of that block. See section 11.4.2 [State for NC], page 169, for a detailed description. This will allow a (future) optimizer to re-order command blocks without worrying about the state changes induced by some of them. Part of the machine state is the cutting tool (see section 11.4.1 [Tools for NC], page 166).

### 11.4.1 Tools for NC

A number of tool types can be defined. At the moment, the package ignores everything about a tool except its diameter and tip radius, and only end mills are properly handled at all. Three types of tools, end mills, drills, and taps, can currently be defined.

The functions for creating tools are somewhat different from the ones used in most of the Alpha\_1 system. Since there are a large number of fields for each tool, it may be cumbersome to specify them all each time. These creation functions allow just a subset of the parameters to be specified (in any order), with the other fields filled in with default values. The slight penalty for this flexibility is that a symbol must accompany each value to indicate for which parameter the value is to be used. In the descriptions which follow, the possible keywords are described, with datatypes for the values indicated as usual, and default values for the fields given in parentheses following the datatype.

```
makeEndMill( KeyValue, ... )
```

**Returns**        <endMill> An object representing an end mill with fields initialized as specified.

**KeyValue**        <keyValuePair> The symbols recognized are:

*Diameter*        <float> (0.0) Diameter of the cutting portion of the tool.

*EndRadius*

                  <float> (Nil) Radius of a ball-end cutter tip or Nil for a flat-end cutter. The value T may also be used to specify that the end radius is 1/2 of the tool diameter.

**CutterLength** <float> (0.0) Length of the cutting portion of the tool (including tip).

**ShankLength** <float> (0.0) Length of the shank (non-cutting) portion of the tool.

**Flutes** <integer> (Nil) Number of flutes.

**Material** <any>(Nil) Material from which tool is made. This will normally be an identifier.

**CenterCutP** <boolean> (Nil) Value is T if the tool can plunge cut, and Nil otherwise.

**ToolSlot** <integer> (Nil) Tool changer slot number for this tool. Will be set by the procedure `ncAssignToolSlot` if Nil.

**makeDrill( KeyValue, ... )**

**Returns** <drill> An object representing a drill, with fields initialized as specified.

**KeyValue** <keyValuePair> The symbols recognized are:

**Diameter** <float> (0.0) Diameter of the cutting portion of the tool.

**CutterLength** <float> (0.0) Length of the cutting portion of the tool (including tip).

**ShankLength** <float> (0.0) Length of the shank (non-cutting) portion of the tool.

**Flutes** <integer> (Nil) Number of flutes.

**Material** <any>(Nil) Material from which tool is made. Will normally be an identifier.

**ToolSlot** <integer> (Nil) Tool changer slot number for this tool. Will be set by the procedure `ncAssignToolSlot` if Nil.

**MakeTap( KeyValue, ... )**

**Returns** <tap> An object representing a tapping tool, with fields initialized as specified.

**KeyValue** <keyValuePair> The symbols recognized are:

**Diameter** <float> (0.0) Diameter of the cutting portion of the tool.

**Threads** <float> (0.0) Thread density measurement, normally in threads per inch.

**ThreadUnit** <keyword> ('INCH) Identifier specifying the unit for measurement of thread density.

**CutterLength** <float> (0.0) Length of the cutting portion of the tool (including tip).

**ChipDir** <keyword> ('DOWN) Identifier specifying the direction of chip ejection, one of 'DOWN or 'UP.

**TapType** <keyword> (Nil) Identifier indicating type of tap (e.g. 'BOTTOMING).

**Material** <any> (Nil) Material from which tool is made. Will normally be an identifier.

**ToolSlot** <integer> (Nil) Tool changer slot number for this tool. Will be set by the procedure **ncAssignToolSlot** if Nil.

Normally, you will create a tool using one of the above procedures, and then assign it a slot in the tool changer with **ncAssignToolSlot**. Although the *CutterLength* field is not used by the NC generation code, it is used when displaying a tool, so you may want to set it. For example,

```
BigCutter := makeEndMill( diameter 0.5, endRadius T, cutterLength 1.0,
                        toolslot 10 )$
ncAssignToolSlot( BigCutter );
```

This creates an end mill tool with a diameter of 0.5, a tip radius of 0.25, and a cutter length of 1.0. The tool is assigned to slot 10 in the tool changer. If another tool had already been assigned slot 10, its assignment would be overridden. If an explicit toolslot is not given when the tool is created, it will be assigned the first available slot the first time it is used (or if **ncAssignToolSlot** is called on the tool).

A tool position object is used to place a tool at a particular position.

**makeToolPosition( KeyValue )**

**Returns** <toolPosition> A tool position object as specified.

**KeyValue** <keyValuePair> The symbols recognized are:

**Tool** <tool> The tool to be positioned.

**TipPosition** <point> (**Origin**) Position of the tool tip in 3-space.

**Orientation** <vector> (Nil) Orientation of the tool; the orientation of the center line of the tool, points from the tip position into the tool. If Nil, the tool is oriented parallel to Z axis. This parameter is only needed for 5-axis milling.

**ChipVector** <vector> (Nil) A vector specifying the direction in which chips should be "thrown". It is used to disambiguate the tool contact point for 5-axis machining. The vector will be projected onto the plane perpendicular to the orientation. If Nil, chips may be thrown in any direction.

**ContactToTip** <transform> (Nil) A transformation, in the coordinate frame with its Z axis along the orientation vector and its Y axis along the chip vector, that takes the tool contact point to the tool tip point. For ball-end milling, this will typically be a rotation about a line parallel to the Y axis. For flat-end milling, it will typically be a translation in X. This parameter is only needed for 5-axis machining.

**SurfaceUV** <VectorOfNumber> (Nil) If this tool position was derived from a surface, the (U,V) surface parameters of the contact point should be saved here as a 2-element vector. This information may be used by restriction region procedures described below.



Tools may be displayed, although it is probably more useful to display *ToolPosition* objects. The display representation for tools is normally a “sketch” of the tool shape, and is suitable for quick display update. To get a more accurate display representation, suitable for shaded rendering, set the *DisplayPrecision* field of the tool to 1.

```
BigCutter->DisplayPrecision := 1;
```

### 11.4.2 State for NC

Each NC operation has an associated state vector. The state vector describes the machine state that should be in effect when the operation is performed. The information contained in the state vector includes the tool, fixture offset selection, feed rate and spindle speed. When the final NC “tape” is generated, the appropriate instructions will be issued to make sure that the correct state is in effect before the operation is executed. A state vector can be created by the routine **makeNcStateVec**.

```
makeNcStateVec( KeyValue, ... )
```

**Returns** <ncStateVec> A newly created state vector containing the specified information.

**KeyValue** <keyValuePair> The symbols recognized are:

**Tool** <tool> (Nil) The tool with which operations should be performed.

**Offset** <integer> (0) The fixture offset to use.

**FeedState** <integer> (0) The type of feed to use. It will be one of 0, 1, or 2 for 'TRAVERSE', 'CONTOUR', or 'CIRCLE' respectively.

**CornerType** <keyword> ('SOFT) When contouring, this specifies the type of corners to produce, and is either 'SOFT or 'HARD. Specification of hard corners will cause a slight dwell at each position in a contour.

**Feed** <float> (Nil) Feed rate to use for the operation. Usually, a value of Nil for this parameter, indicating that no feed rate is specified, will cause an error when contouring.

**Speed** <float> (Nil) Spindle speed to use. It should be specified explicitly, the default Nil will usually cause an error when used.

**CircleDir** <keyword> 'CW Direction in which circles should be cut. It should be either 'CW or 'CCW.

**CirclePlane** <keyword> ('XY) Plane in which circular motion will be produced. It should be one of 'XY, 'YZ, or 'XZ.

**Inches** <boolean> (T) Whether units are measured in inches (value T) or millimeters (value Nil).

**Absolute** <boolean> (T) Indicates whether positional information is specified absolutely or relative to the previous position. The current contouring code does not deal properly with relative positioning. (I.e., don't specify Nil unless you are doing your own position generation.)

Here is an example of using **makeNcStateVec** to create a state vector using the end mill created above, with a feed rate of 8 inches per minute and a spindle speed of 3500. All other parameters are given their default values.

```
BigState := makeNcStateVec( Tool BigCutter, Feed 8, Speed 3500 );
```

Frequently, you may wish to create a base state for a machining sequence, and then modify it slightly for each step (by changing the tool, feed, and speed, for example). The modification can be accomplished by creating a copy of the original state vector and then changing some of the fields. A state vector is copied with the function **ncCopyState**.

```
ncCopyState( State )
```

**Returns**     <ncStateVec> A copy of the given state vector.

**State**        <ncStateVec> State vector to be copied.

In the next example, a base state is created for contour cutting with fixture offset of 12. Then, two modified state vectors are created to use the BigCutter, one with a slow feed rate and one with a faster feed rate.

```
DefState := makeNcStateVec( FeedState 'Contour, Offset 12 );
SlowBigState := ncCopyState( DefState );
SlowBigState->Tool := BigTool
SlowBigState->Feed := 4;
SlowBigState->Speed := 3500;
BigState := ncCopyState( SlowBigState );
BigState->Feed := 12;
```

In most of the contexts in which a state vector is used, the vector is not copied. Thus, any changes made to the contents of a state vector will usually be reflected in all the places it is used (but not in NC tape code that has already been generated). You should not count on this, though, and you should be careful to copy a state vector with **ncCopyState** if you want to make one similar to it.

### 11.4.3 Generating NC tapes

A NC "tape" file is created by first performing a few setup steps, then executing commands that generate machining codes, and finally writing the "tape" to a disk file. Before issuing machining commands, **ncNewTape** and **ncStateInit** must be called.

The **ncNewTape** function sets up program state to generate a new NC tape file. It creates a new tape data structure, and outputs a "program header" tape block with the name and title.

```
ncNewTape( Name, Title )
```

**Returns**     <Nil> Get ready to generate a new NC tape file.

**Name**        <string | integer> The tape name, usually used by the NC machine to identify this tape after it has been downloaded.

**Title**       <string> Descriptive title for tape, which may show up in NC machine directory listing.

There are some global variables that are used by the NC tape generation routines. These are (with initial value in parentheses):

**NcBlockInc!\***

<integer> (10) It specifies the increment applied to the tape block number from one block to the next. The default value is 10 (set by **ncNewTape**). It may be changed at any time after calling **ncNewTape**. If your NC tape is going to be longer than 1000 steps, you may want to set this variable to 1.

**NcRounding!\***

**<number>** (10000.0) Specifies the amount of rounding that should be performed on positional values. Rounding is performed by multiplying by **NcRounding!\***, rounding to the nearest integer, then dividing by **NcRounding!\***. The value should generally be a power of 10.

The **ncStateInit** function initializes all the state variables associated with the NC code generation. It should be called at the beginning of a tape, and whenever the current machine state is unknown.

**ncStateInit()**

**Returns** **<Nil>** Initialize all the state variables for NC code generation.

The basic tape code generating routine is **ncGen**. Its argument is an object representing a tape block, and it adds code to the NC tape data structure to execute the operation(s) implied by the tape block object (including setting the associated state). The individual tape block objects are described in the next section. After the tape has been generated, it must be written into a file using **ncFile**.

**ncGen( TapeBlock )**

**Returns** **<list>** A list of strings, the actual machine code generated for the block.

**TapeBlock**

**<tapeBlock>** A tape block object specifying the desired state and operation to be performed.

**ncFile( FileName )**

**Returns** **<string>** The given file name.

**FileName** **<string>** The name of the file into which the tape will be written.

#### 11.4.4 NC Tape Block Objects

Each tape block object contains the parameters associated with a single machining or setup operation. All tape block objects contain a state vector (see section 11.4.2 [State for NC], page 169).

The most general is the **ncControl** object. It contains a single string or list specifying arbitrary "G-codes". It is the only tape block object with no associated state.

**ncControl( GCodes )**

**Returns** **<ncControl>** An object containing the specified "G codes."

**GCodes** **<string | list>** A single string, or a list of strings, IDs and pairs specifying the desired G code actions.

Examples of the use of **ncControl** for driving (at least) the Monarch VMC45:

```
ncGen ncControl "M03 M08"; % start spindle and flood
ncGen ncControl list( 'M . 3, 'M . 8 ); % another way to do above
ncGen ncControl( "g00 g90 x0 y0 m05 m06 m09 t40 ! ( shut down )" );
ncGen ncControl( "m30" ); % end of tape
```

A tape should be started with a "safe block." This makes sure that the machine state will be totally specified at that point, and prevents accidentally "inheriting" state from the previous run on the machine. The **ncSafeBlock** object contains only a state vector, and will cause generation of a safe block by **ncGen**.

**ncSafeBlock( State )**

**Returns** <ncSafeBlock> An object containing the given state vector.  
**State** <ncStateVec> Desired state vector.

For example, to generate a safe block using the *BigState* derived above, one would say

```
ncGen ncSafeBlock BigState;
```

The basic machining operation is to move the tool to a certain position. The **ncPosition** object provides this capability. It may be created using **makeNcPosition**.

```
makeNcPosition( KeyValue, ... )
```

**Returns** <ncPosition> An ncPosition object.  
**KeyValue** <keyValuePair> The symbols recognized are:

<b>State</b>	<ncStateVec> (Nil) Desired machine state for this operation. Must be specified, Nil causes an error.
<b>!3Axis</b>	<e3Pt> (Nil) 3D point specifying position of tool tip for 3 axis motion.
<b>RotAxis</b>	<r2Vec> (Nil) 2D vector specifying rotational positioning for 5 axis motion. The first component is A axis, and the second is B axis. If Nil, 3 axis motion is performed. <i>This parameter is currently ignored.</i>
<b>Center</b>	<point> (Nil) 3D point specifying center of circle for circular motion. Only the components selected by the <i>CirclePlane</i> field of the state vector are used.

You may build up a list of NC block objects and output them in sequence by calling **ncGenPath**.

```
ncGenPath( TapeBlockList )
```

**Returns** <Nil>  
**TapeBlockList**  
 <list> A list of NC tape block objects or tape block lists. NC code is output for each block in sequence.

### 11.4.5 Surface Contouring With NC

A fairly powerful surface contouring facility is available to generate NC code for cutting arbitrary spline surfaces. The cutter is moved along isoparametric curves of constant V value. The spacing of points on the curves, and the spacing of the curves may be determined in a number of different ways.

The movement of the cutter may be restricted by supplying a restriction specification. This feature may be useful for preventing gouging or for milling an arbitrary region of a surface.

The two main procedures for surface contour milling are **ncSrfIso** and **ncSrfIsoZag**. The only difference is that **ncSrfIso** generates all its isoparametric curves with the cutter moving in the direction of increasing U, while **ncSrfIsoZag** alternates the cutter direction without picking up the cutter. Thus, NC tapes generated by **ncSrfIsoZag** will run somewhat faster, but may have rougher scalloping due to the changing direction of milling. The arguments to both functions are the same, so only **ncSrfIso** will be described in detail.

```
ncSrfIso( Srf, SrfSpec, CrvSpec, NcState, Restriction )
```

**Returns** <Nil> Generate isoparametric curves as cutter path.

<i>Srf</i>	<surface> The surface to be milled.
<i>SrfSpec</i>	<integer   listOf number   float> Specifies the number of isoparametric curves to be cut. If an integer is given, then exactly that many curves will be cut, equally spaced in V. If a list of numbers is given, then a curve will be cut with V equal to each value in turn. If a floating point number is given, then a list of V values will be computed so that the cutter paths will be no further apart than that along the U=0 edge of the surface. The function <b>ncChordParms</b> (see below) generates this list.
<i>CrvSpec</i>	<integer   listOf number   float> Specifies the number of points to be generated along each isoparametric curve. Values may be as for <i>SrfSpec</i> , with the exception that U values are generated. If <i>CrvSpec</i> is a floating point number, then a list of U values is computed so that the cutter positions along the V=0 edge of the surface are no further apart than that distance.
<i>NcState</i>	<ncStateVec> The desired NC machine state. Some other states will be derived from this, as specified by some global variables (described below). The <i>Tool</i> and <i>Feed</i> elements of the state vector must not be Nil.
<i>Restriction</i>	<function   Nil> A function that is applied to each isoparametric path (or to the entire path, for zig-zag cutting) to restrict the possible cutter positions. Specify Nil for no restriction.

The action of **ncSrfIso** relies on the value of two global variables. These control the speed of the cutter on the initial cut and the height to which the cutter will retract when moving from one point to another without cutting (traversing).

#### **NcSlowFeed!\***

<number> The feed rate used for the first cut. The assumption is that the cutter will be cutting full width on the first cut, and must therefore move slower. All other cuts are assumed to be partial width, and are performed at the feed rate in the state vector provided as argument.

#### **NcSafeZ!\***

<number> When the cutter must be moved from one position to another without cutting, it will first be retracted to this Z value.

Sometimes, the two easy options provided by **ncSrfIso** specifying the number of curves or the number of points along a curve (i.e., an integer or a floating point number) do not provide good spacing. The function **ncChordParms** will compute a list of parameter values for any curve so that the corresponding points are not further apart than a given distance.

#### **ncChordParms( Crv, Spacing )**

*Returns* <listOf number> List of parameter values so that the corresponding points on the curve are no further apart than the specified spacing.

*Crv* <curve> Curve to compute parameter values for.

*Spacing* <number> Desired maximum spacing.

You may be able to use this to generate an appropriate surface or curve specification for **ncSrfIso** by applying it to a curve extracted from the surface. For example, to compute surface and curve specifications based on the curves at the parameter values V=0.5, U=1.0, you would use the following statements:

```
% Extract U and V curves from surface Srf.
```

```

UCrv := crvInSrf( Srf, ROW, 1.0 );
VCrv := crvInSrf( Srf, COL, 0.5 );
% Get parameter lists for point spacing of 0.1
% and curve spacing of 0.02.
CrvSpec := ncChordParms( UCrv, 0.1 );
SrfSpec := ncChordParms( VCrv, 0.02 );
% Now mill the surface
ncSrfIso( Srf, SrfSpec, CrvSpec, BigState, nil );

```

It is also possible to move the cutter along a single curve in space by calling **ncCrvMill**.

```
ncCrvMill( Crv, Spec, Contact, Depth, DoCut, NcState )
```

<b>Returns</b>	<group> A group object containing all the generated tool positions.
<b>Crv</b>	<curve> The curve to be milled.
<b>Spec</b>	<integer   listOf number   float> Specification of parameter values for milling. As in <b>ncSrfIso</b> , above.
<b>Contact</b>	<number   transform> If a number, specifies the horizontal distance from the tool tip to the point on the curve. Measured perpendicular to the curve, with positive direction to the left when walking along the curve in the direction of increasing parameter. If a transform, specifies the transformation from the tool contact point to the tool endpoint in the coordinate system with the Z axis parallel to the curve tangent vector, the X axis perpendicular to the curve in the horizontal plane (oriented to the left, as above), and the Y axis perpendicular to both X and Z (pointing up), with the origin at the contact point.
<b>Depth</b>	<number> If non-nil, specifies that the tool tip should be at a constant Z depth (equal to the given value). In this case, only the X-Y motion of the tool will be affected by the curve.
<b>DoCut</b>	<boolean> If non-nil, NC code will be generated.
<b>NcState</b>	<ncStateVec> Desired machine state for this operation.

### 11.4.6 Profile NC generation

A common machining operation is to cut around the outside or inside of a profile curve, which is made up of circular arcs and straight lines. It would be overkill to use a general B-spline method, such as provided by **ncCrvMill** for this purpose, and the result would not be as good as is possible, since **ncCrvMill** approximates all curves (including circular arcs) by straight line segments.

The function **ncGenProfile** provides way to cut profile curves. It moves the cutter along a curve offset by the cutter radius from the given profile curve. It also has a provision for doing an initial rough cut.

```
ncGenProfile( Profile, ClimbMill, Depth, RoughOffset, NcState )
```

<b>Returns</b>	<Nil> Generates NC cutter path for a profile curve.
<b>Profile</b>	<profile> A profile object constructed from arcs, straight line segments or points.
<b>ClimbMill</b>	<boolean> If Nil, the cutter will be placed to the right of the profile, otherwise it will be placed to the left.
<b>Depth</b>	<number> Specifies the depth of the cut as a Z coordinate.

*RoughOffset*

<number> Distance the rough cut should be from the final desired curve.

*NcState* <ncStateVec> Desired machine state for this operation.

The profile offset generation procedure tries to be intelligent about corners. "Inside" arcs that are smaller than the offset distance turn into sharp corners, but there are cases in which a large offset distance will cause the wrong curve to be generated.

### 11.4.7 Pocketing for NC

The **nc2DPocket** procedure generates N/C machine commands to mill a two-dimensional pocket at a constant depth. The two-dimensional pocket is expected to be a multiply-connected region parallel to the XY plane of the machine coordinate system. The outermost boundary of the pocket is called the boundary of the pocket, while all the other bounding curves inside the boundary of the pocket will be called boundaries of the islands.

**ncCmds2d**( *BdryInfo*, *ToolPathInfo*, *NcMachInfo*, *NcState*)

*Returns* <Nil> Generates NC cutter path for a pocket.

*BdryInfo* <listOf curve> The list of curves which bound the pocket. First curve in the list should be the boundary of the pocket. The rest of the curves, if any, are the boundaries of the islands. The curves should be oriented such that when we walk through the curves along increasing parametric values, the curves form a multiply-connected region, and the region is to our left. Only curves with open end conditions and coincident end-points are allowed. <...> ...

*ToolPathInfo*

<listOf pairs> A list of keyword-value pairs with information about toolpath generation.

'CutDepth

<number> The z coordinate of the bottom of the pocket. Default value is -0.25.

'SideStepFun

<keyword> The name of a function which should take as a single parameter the distance from the tool position to its closest boundary. It should return the amount of side-step of the tool. If the value returned by the function is larger than the radius of the tool, part of the pocket may be left uncut. The default is a constant side-step function with side-step equal to  $0.8 * \text{ToolRadius}$ .

'Border

<number> This value specifies the width of the border around the inside of the boundary of the pocket and the outside of each islands which will remain uncut. No negative value is allowed. Default value is 0.0 (no border).

*NcMachInfo*

A list of keyword-value pairs with information about N/C machine command generation.

'NcSlowFeedFun

<keyword> This function should return the machine feedrate as a function of the amount of side-step in each cut. The default value is a function which generates feedrates suitable for cutting wax.

#### 'MoveDownFRt

The Feedrate to plunge tool into stock. The default is 15 ipm.

**NcState** <ncStateVec> An NcStateVec object which represents the state of current cutting. It will be modified to the state of cutting at the end of pocket milling on returning from this function.

The global variable **CLPathName!\*** is used by the pocketing routine.

#### CLPathName!\*

<string> The name of the scratch file used by **nc2DPocket**. If the value of this variable is Nil, file ".tmp.cl" will be used.

In addition to this global variable, all those affecting NC command generation in the basic NC package will be effective too. Note especially the value of **NcSafeZ!\***.

The N/C machine commands generated for the pocket milling will be appended to the existing N/C command buffer.

The object coordinate system of the pocket should be oriented so that the pocket is parallel to the XY plane and dents into the negative Z direction. The origin of the object coordinate system will be made coincident with the origin of the NC machine, while the X, Y and Z axes of the object coordinate system coincide with the directions of the X, Y and Z axes of the NC machine, respectively.

The milling tool is directed to clean out each of the subpockets sequentially. Within each subpocket, the tool starts from the innermost position, and is directed to go toward the boundary through spiral type of paths until all the area within the subpocket is cleaned. Then, the tool traverses on the Z plane at **NcSafeZ!\*** to one of the uncleaned subpockets. The above process repeats until all the area within the pocket is cleaned. No optimization of the sequence of cleaning of the subpockets is made. When the tool enters a subpocket, it simply plunges into the workpiece from above at the innermost position of the subpocket. There is no option to drill the positions where the mill plunges into the workpiece before milling.

The tool will follow the control polygon of the curves in the *BdryInfo* parameter. No refinement of the curves is done by **nc2DPocket**. A certain amount of refinement is expected to have been done on the curves by the user. Some caution is required, as numerical instabilities may occur when the curves are over-refined.

## 11.4.8 Simulating NC

There is not much help provided for simulation currently. You can display groups of tool positions, but that's about all. For profiles, you can compute the offset curve yourself using **ncProfileOffset** and show it.

**ncProfileOffset**( *Profile*, *ClimbMill*, *Offset* )

**Returns** <profile> A new profile, offset from the original according to the other arguments.



*ClimbMill* <boolean> If Nil, offset to the right of the profile, otherwise offset to the left.  
*Offset* <number> Offset distance.

## 11.5 Finite Element Analysis

This section provides a typical scenario of a structural finite element analysis using *shape\_edit* and the *adina* analysis code. Currently, *adina* is the only package to which Alpha\_1 has any finite element interface at all.

Generally, the entire analysis process consists of two cycles. The first cycle starts with a pre-process for generation of an initial, uniform, relatively coarse mesh generation. The initial mesh may be displayed. The input data for the analysis can be generated after the mesh is visually examined. After executing the analysis, the analysis result is interpreted, evaluated and attached back to the initial mesh. This creates criterion surfaces, on which further mesh generation can be based. The results can be displayed, using a color variation scheme for strain energy distribution, or with transparent color for shape distortion. As a special case for the planar domains, the Z-coordinates (originally 0's) are replaced with the nodal strain energy value, so the criterion surfaces can be visually examined using standard 3D graphics techniques. The first cycle may be repeated for different dimension specifications of the initial, uniform mesh. The second cycle starts with the subdivision of the criterion surfaces according to the subdivision criterion. Thus a synthesized mesh is obtained and the input data for the finite element analysis is re-generated. Re-executing the analysis upon this synthesized mesh, the result is evaluated and compared to that of the initial, uniform mesh. Again, the criterion surfaces can be constructed by interpreting the analysis result and attaching them back to the near-optimum mesh. The criterion surfaces for the planar domains can be visually examined, again treated as a special case. The second cycle may be repeated for different criteria for subdividing the criterion surfaces obtained from the same initial, uniform mesh.

The pre-processing consists of three major steps: defining the domain topology, breaking down this topology into regions and specifying all the related attributes for each region. Afterwards, a procedure is called to process these regions into a "domain" which contains a two-dimensional uniform mesh with complete nodal, elemental, and loading data. This data can be further processed by a procedure that produces an input file for *adina*.

For a two-dimensional domain, the topology is determined by its contours. This is usually achieved by constructing a collection of curves (lines) such that it constitutes closed areas on which the analysis is performed. For different physical meanings, the user may have to describe the domain in terms of several closed areas. But since it is only the domain contours that are specified, no surfaces need to be constructed at this point.

Next, a "zoning process" is undertaken to form surfaces out of those boundary curves. There are several reasons for not treating the entire domain as one single surface. The analysis domain may consist of more than one material and hence be modelled by different elasticity moduli. A second consideration is that during this process the analysis domain is represented in terms of B-spline tensor-product surfaces which must be (parametrically) rectangular. Forcing the entire domain into parametrically rectangular in some case may produce ill-shaped elements and the resulting accuracy will be severely compromised. Therefore, it is suggested that the entire domain be represented by a collection of (geometrically) almost-rectangular surfaces. Furthermore, these surfaces can be constructed by determining their four boundary curves and applying the *boolSum* construction operator. If *boolSum* is used, each such surface must be convex. Naturally the domain may not be convex and thus may have to be broken down into a composition of convex surfaces. The user may decompose the domain however it is required with respect to these considerations except that

there are three important things that have to be kept in mind. First, any shared boundary curve must be by virtue of one construction and hence of one single parametrization. Second, no surface is allowed to be adjacent to more than one surface along one of its four boundary curves. Third, all the loading information and displacement constraints are to be specified only at either corners or the boundaries of the surface. Therefore during this process, the user might have to replace the domain contours specified in the previous session by a collection of smaller boundary curves and might have to construct a few more boundary curves from which surfaces are formed.

Finally, all the surfaces are to be associated with non-geometric attributes and made into regions. For each region, the following attributes may be attached if desired.

*Boundary curve identifications*

Each region should have four boundary curves, and four such system-generated identifications will be recorded.

*Region adjacency relations*

Each region will be given a system-generated identification. Theoretically each region has one or more neighboring regions and the adjacency relation can be represented by recording identifications of all its adjacent regions.

*Dimension specifications*

Each region is (parametrically) subdivided into an N by M uniform mesh according to the user's specification.

*Material properties*

The user must provide Young's modulus and Poisson's ratio for each region and/or thickness and density for each region.

*Loading information*

The region may have either concentrated loads (on its corner points) or distributed loads (along its boundary curves).

*Displacement constraints*

If the region is subject to some displacement constraints, they can be specified point-wise or boundary-wise.

At this point, the user has completed the problem specification. The result is a list of regions which can be further processed into an input file for the analysis.

For the remainder of this section, a roller bearing will be used as an example. The geometry specifications described above are done in a file called "Roller.r". The rest of the execution process is contained in an execution file called "execRoller.r". The geometry specification can be loaded with

```
in "Roller.r";
```

This builds the initial specification, and results in a list of regions. In order to access the analysis functions, it will be necessary to load the appropriate finite element module into `shape_edit`. There are separate 2D and 3D load modules, and it is recommended that you only load the one you need.

```
load "fem2dlib";
load "fem3dlib";
```

The list of regions can be converted into a domain data structure using `domain2DUniform`. From the domain, an initial uniform mesh is generated with `femUniformMesh`. Before the input data for the analysis is generated, we might want to examine the mesh visually.

```
OriginalDomain := domain2DUniform( SomeRegionList )$
SomeMesh := femUniformMesh( OriginalDomain )$
SomeMeshDisplay ^= group SomeMesh$
```

If the mesh appears to be acceptable, we want to generate the input data for this domain. We specify the element type, the header line and the input file name, and call the input file generator, **femInput**. Users should adopt some convenient naming scheme for the files produced. In this example, we use prefix "MP" for "Multiple Patches", "R" for roller and "8" for dimension 8 by 8. We use ".data" to indicate that it is an input data file.

```
OriginalDomain->elementType := 'STRESS;
Header := "Roller bearing (MP 8 by 8), Jonathan Yen, April 30, 1985"$
FileName := "MPR8.data";
femInput( OriginalDomain, Header, FileName );
```

This "MPR8.data" file is to be used for analysis by executing the command:

```
adina < MPR8.data > MPR8.out
```

(At Utah, **adina** is on director "/usr/src/local/adina/work".)

This "MPR8.out" file contains all the detailed input card images as well as the output as nodal displacement and stress values. To shorten the postprocessing, it is suggested that a text editor (emacs) be used to remove everything from the beginning of the file until the line containing "D I S P L A C E M E N T", and store the result in a ".output" file.

For output interpretation, specify the output file name and call the output interpreter, **femOutput**. The output interpretation routine returns a floating number which is the highest strain energy obtained from the original domain. It is important to capture this value since it will be used later as a reference for further subdivision. It provides a good heuristic as to how far the original mesh is to be refined. It is suggested that a fraction of this value, (e.g., 1/2, 1/3, 1/4, 1/8, 1/10), be used for the subdivision criterion.

```
FileName := "MPR8.output";
Highest := femOutput( OriginalDomain, FileName );
```

Now the original domain has been loaded with nodal results, so we call the result evaluator, **femEvaluation**. The total number of nodes, the total number of elements, maximal normal stress values and shear stress values, total number of degrees of freedom, maximal axial nodal displacements, and other information will be printed.

```
femEvaluation( OriginalDomain );
```

To simulate the shape deformation, the **femDisp** routine for extracting the nodal displacements and constructing a distorted finite element mesh which is used as the control mesh of the distorted domain geometry. By transforming this one quarter of the domain, the entire distorted domain is obtained and dumped.

```
Distorted := femDisp( OriginalDomain, 400000 )$
Half := append( Distorted, objTransform( Distorted, ry( 180 ) ) )$
Whole := append( Half, objTransform( Half, rx( 180 ) ) )$
Distorted := objTransform( Whole, tz( 1.0 ) )$
dumpA1File( Distorted, "distorted.a1" );
```

The criterion surface can be constructed and attached to each region using **femCriterion**.

```
femCriterion( OriginalDomain );
```

As indicated above, treating the planar domain as a special case, we can visualize its criterion surfaces (with **showCriterionSrf2D**) by properly scaling the energy term making it more compatible to the size of domain surface. The choice of 5.0 is simply because the Roller is 5.0 by 5.0 units.

```
ScalingFactor := 5.0 / Highest;
```

```

OriCriSrf := showCriterionSrf2D( ScalingFactor, OriginalDomain )$
OriCriSrfDisplay ^= group OriCriSrf$

```

To render the criterion surfaces with the color variation mechanism, first dump these criterion surfaces using `dumpCriterionSrfFile`. This generates a standard Alpha\_1 text file. This ".a1" file, when preceded with a header file "rollerEnergy.hdr" which specifies the color coordinate and color map ("rollerEnergy.cmap"), can be rendered by the render program.

```

dumpCriterionSrfFile( OriginalDomain, "MPR8.a1" );

```

This is the end of the first cycle. We may repeat it with different dimension specifications for the uniform domain.

If we follow the same example, this marks the beginning of the second cycle. It is important to make provision for possible repetition of the second cycle since the refinement of the original mesh is done destructively. Hence we make a copy of the original domain and carry out the refinement on this copy. Let's first try with the subdivision criterion as half of the highest strain energy value obtained from the original mesh.

```

SomeDomain := totalCopy OriginalDomain$
SomeDomain->SubdivisionCriterion := Highest / 2;

```

Using the procedure `domainNearOptimum`, the domain is turned into a new domain with an entirely different mesh configuration.

```

NewDomain := domainNearOptimum( SomeDomain )$

```

We then repeat the input preparation process, this time with a different header line. The naming convention used here is that "N" is for near-optimum and "2" indicates it is half of the highest value in the previous run.

```

Header :=
  "Roller bearing (MP Near Optimum 1/2), Jonathan Yen, April 25, 1985"$
FileName := "MPRN2.data";
femInput( NewDomain, Header, FileName );

```

The rest is similar to the post-processing in the first cycle.

```

FileName := "MPRN2.output";
NewHighest := femOutput( NewDomain, FileName );

femEvaluation( NewDomain );

femCriterionSrf( NewDomain );

ScalingFactor := 5.0 / NewHighest;
NewCriSrf := showCriterionSrf2D( ScalingFactor, SomeDomain )$
NewCriSrfDisplay ^= group NewCriSrf$

dumpCriterionSrfFile( NewDomain, "MPRN2.a1" );

```

If we want to repeat the second cycle for different subdivision criteria, go back to the beginning of the second cycle, change the choice of the subdivision criterion (1/4, 1/8?) and change the file names accordingly.



## 12. Leaving Shape\_edit

This section describes the preparation of output from **shape\_edit** for use in the other system utilities not available from **shape\_edit**.

### 12.1 Shells

If the data which has been created in **shape\_edit** is to be combined using the boolean expression combiner program (see chapter 14 [Combining Objects], page 191), it will probably be necessary to group surfaces into aggregate objects called *shells*.

**shell**( *SurfaceList*, ... )

**Returns**    <shell> A shell object constructed from the given surfaces.

*SurfaceList* ...

<surface | listOf surface> The surfaces which will make up the shell.

### 12.2 Attributes

Attributes can be associated with objects, and are formatted by **dumpA1File** (see section 12.4 [Saving Data], page 185) into the text file representation of objects in an object stream. All of the non-geometric properties of objects are controlled by attributes, as well as some quasi-geometric properties such as adjacencies between surfaces.

One group of functions manipulates attributes associated with objects. Each attribute of an object has a name and a value.

To set or change an attribute value of an object, use **setAttr**. An attribute can be removed from an object using **remAttr**. To find the value of an attribute, use **getAttr**.

**setAttr**( *Obj*, *AttrName*, *Value* )

**Returns**    <Nil> Sets an attribute of the object.

*Obj*        <object> The object to which the attribute belongs.

*AttrName*   <string | keyword> Name of the attribute.

*Value*      <any> Value of the attribute.

**remAttr**( *Obj*, *AttrName* )

**Returns**    <Nil> Removes an attribute of the object.

*Obj*        <object> The object to which the attribute belongs.

*AttrName*   <string | keyword> Name of the attribute to remove.

**getAttr**( *Obj*, *AttrName* )

**Returns**    <attribute> Get a particular attribute of an object.

*Obj*        <object> The object to which the attribute belongs.

*AttrName*   <string | keyword> The name of the attribute to retrieve.

Attributes may also be created without assigning them to a particular object. The primary reason for doing this is to use them as "global" attributes. "Global" attributes may occur at the top level of an object stream (file), or exist as independent objects in **shape\_edit**, rather than being embedded in an object. Global attributes are not currently used within the **shape\_edit** environment. In



the object stream, where applicable, they set default values for objects in the stream that follows. Thus, it is common practice to collect attributes such as color and rendering resolution in a separate header file, rather than embedding them in individual objects. It is much easier to change a color in one place than on every surface in a file.

Global attributes as independent objects are created with **attribute**, and their name and value fields accessed using **attributeName** and **attributeValue**.

**attribute( Name, Value )**

**Returns** <attribute> Construct an attribute with the given name and value.

**Name** <string | keyword> The name of the attribute.

**Value** <any> The value of the attribute.

**attributeName( Attr )**

**Returns** <string | keyword> The name of the attribute object.

**Attr** <attribute> The attribute object.

**attributeValue( Attr )**

**Returns** <any> The value of the attribute object.

**Attr** <attribute> The attribute object.

### Built-in Attributes

All attributes have names and values. Some names ("resolution", "width", "color", "backColor" and "opacity") are built-in attributes, known to the Alpha\_1 object reader which interprets an object stream. Most of the utility programs which read object streams will use global built-in attributes as defaults for objects in the stream which don't specifically set a built-in attribute. This is a convention, however, and is subject to subtle variations from one program to the next.

Built-in attribute values can have either a numerical value, or an attribute instance name in place of a value. Numerical color values are lists of 3 numbers, giving red, green, and blue intensities. Other numerical values are single numbers.

Instance names refer to attribute instances external to the object, which may be set globally in a separate prefix file early in the object input stream. Instance names can be given as identifiers or strings. For identifiers, the name will be completely upper-cased in the output file.

Some examples are:

```
setAttr( HandleSrf, "name", "Handle" );
setAttr( HandleSrf, "color", "handle_color" );
setAttr( BodySrf, "color", list( 255, 255, 255 ) );
setAttr( BodySrf, "resolution", 5.0 );
```

### User-Defined Attributes

Any names not known as built-in attributes are user-defined attributes. The value of a user-defined attribute may be a string, a floating point number, or an integer. User-defined attribute names and string attribute values are given as strings in the correct case for output.

Since the type of the attribute is inferred from the value, additional information is necessary when the value is not given directly, but is an instance of a named global attribute object. This is supported by giving the attribute type and value using the **stringAttr**, **intAttr**, and **floatAttr** functions. For example:

```
setAttr( Obj, AttrName, stringAttr( InstanceName ) );
```



```

setAttr( Obj, AttrName, intAttr( InstanceName ) );
setAttr( Obj, AttrName, floatAttr( InstanceName ) );

```

### Adjacency Attributes

Adjacencies between surfaces "weld" the surfaces together into boundary representations of volumetric objects. They are used primarily by the **combine** program to allow intersection curves with other objects to cross surface boundaries during the surface-trimming phase of the volume union, intersection, difference, and cut operations.

Adjacencies are automatically declared for the objects created by some operations, most notably the primitives and rounded primitives. In other cases the adjacencies will have to be added after the surface construction operations. Many of the adjacencies which need to be declared in a group of surfaces can be detected by the **autoMkAdjacent** routine (described below), or adjacencies can be added manually.

To manually declare adjacencies between surfaces use **mkAdjacent**. Adjacency entries are put on the attribute lists of both surfaces by **mkAdjacent**.

```
mkAdjacent( Srf1, Side1, Srf2, Side2 )
```

**Returns** <Nil> Adds adjacency attributes to both surfaces declaring the mutual adjacency.

**Srf1, Srf2** <surface> The two surfaces which have an adjacent edge.

**Side1, Side2**

<keyword> One of 'TOP', 'BOTTOM', 'LEFT', or 'RIGHT' indicating which edges are adjacent.

**Srf1** and **Srf2** are surface objects, each with an **adname** attribute to be used as the surface name for adjacency declaration. In case an **adname** attribute is not already present before the call to **mkAdjacent**, an **adname** attribute will also be deposited.

**Side1** and **Side2** denote the sides of the parametric domain to be joined, using constants:

```

'top      the low U edge, defined by the first row of the mesh.
'bottom   the high U edge, defined by the last row of the mesh.
'left     the low V edge, defined by the first column of the mesh.
'right    the high V edge, defined by the last column of the mesh.

```

An example of declaring a surface adjacency might be

```

setAttr( BodySrf, "name", "body" ); % Set name attributes first.
setAttr( TopSrf, "name", "top" );

% Right edge of top surface is adjacent to left edge of body surface.
mkAdjacent( BodySrf, 'left, TopSrf, 'right );

```

A surface may be adjacent to itself (think of rolling a piece of paper into a cylinder):

```

% Surface has adjacent left and right edges.
setAttr( CylinderSrf, "name", "cylinder" );
mkAdjacent( CylinderSrf, 'left, CylinderSrf, 'right );

```

The **autoMkAdjacent** and **autoMkAdjacentL** utilities are for automatically declaring surface adjacencies as an aid to constructing shells for combiner input. They also attempt to ensure that the surfaces have compatible orientations.

```
autoMkAdjacent( Srf1, ... )
```

*Returns*     <Nil> Looks for and declares adjacencies in a set of surfaces.  
*Srf1, ...*     <surface> The surfaces to be checked.

**autoMkAdjacentL( SrfList )**

*Returns*     <Nil> Looks for and declares adjacencies in a list of surfaces.  
*SrfList*     <listOf surface> The surfaces to be checked.

The return value is a list of surfaces with adjacencies declared. None of the input surfaces appear in the output (they are copied). Any necessary **adjname** attributes are added. The orientation of the first surface is imposed on the others; therefore surfaces are reversed as necessary.

The **autoMkAdjacent** function only detects adjacencies along entire boundary curves. For two curves to match, they must have the same order, the knot vectors must be the same (up to fuzz) when normalized over the interval [0,1], and the control points must be the same (up to fuzz). As a result, it is still necessary to declare by hand adjacencies that do not match entire curves. The **autoMkAdjacent** function, for example, would not discover the adjacencies necessary for a cylinder. The current implementation assumes that all pre-declared adjacencies are correct, including having compatible orientation between the two surfaces.

It is an error if it is not possible to pass from any surface patch to another by a chain of adjacencies (either pre-declared or discovered by this procedure). It is also an error if one of the surfaces has a pre-declared adjacency to a surface not included in the input list. If a surface or a collection of surfaces forms a non-orientable surface (e.g., a Mobius band), **autoMkAdjacent** will detect this and signal an error. This diagnostic may also occur if a pre-declared adjacency is incorrect, (making **autoMkAdjacent** think that the surface is non-orientable).

The **autoMkAdjacent** function is intended to be a user-level function. To have any reasonable performance, all system routines should declare adjacencies themselves (rather than calling **autoMkAdjacent**) if at all possible.

There is one function, **getAdjacencyVectors**, which can be used to visually check the declared adjacencies on a set of surfaces. This new feature is used to display adjacencies in **shape\_edit**. When you call **mkAdjacent** or **autoMkAdjacent**, adjacency vectors are created for later display and are saved as an attribute of the surface. The vector will lie somewhere near the middle of the parametric range of the corresponding edge. The base point is on the edge and the end point is in the direction of the partial at the edge, pointing into the surface interior. The location of the vector is randomized to aid in detecting incorrect adjacencies. If the adjacent edges are identical and have the same parametrization, the vectors should have the same base point but different directions. If they don't line up, either the edges are only partially adjacent, they have different parametrizations, the adjacency declaration was incorrect, or any combination of the above. The **getAdjacencyVectors** function just returns the set of vectors, which you can then **show** or **unshow** like any other geometric object.

**getAdjacencyVectors( SrfList )**

*Returns*     <group> A group of vectors representing the adjacencies declared for each edge.  
*SrfList*     <listOf surface> The group of surfaces to be checked.

This will return a group of all the adjacency vectors from the list of surfaces **SrfList** which can then be shown.

More information on adjacencies in the context of the combiner program may be found in the chapter on combining objects (see chapter 14 [Combining Objects], page 191).

## 12.3 Surface Orientation

The orientation of surfaces is not usually critical during the design of objects in the `shape_edit` environment. But orientation is important for correct results in shaded raster rendering and in boolean combination of objects. Orientation issues will need to be considered as objects are being dumped from `shape_edit`. You can use the `setSrfNorms` function (see section 6.3 [Displaying Objects], page 35) to cause surface normals to be displayed during model construction. A more complicated, but very graphic check is to dump the surfaces to a file and then render the data. The default back color is currently a hot pink (while the default front facing color is dull grey). So the resulting image will easily show up surfaces which need to be reversed. It may be necessary to render several views so that all surfaces can be seen.

For further references on orientation, see chapter 5 [Orientation Conventions], page 29, and the sections in the combiner description (see section 14.2 [Preparing a Model for Combining], page 194) and the Rendering Preparation chapter (see chapter 15 [Rendering Preparation], page 203).

## 12.4 Saving Data

The data created in `shape_edit` must be written to a file in order for the C utility programs to operate on it. Currently the data is dumped in Alpha\_1 text format using `dumpA1File`. For more on the text format see [Text File Format], page 299.

`dumpA1File( ObjectList, FileName )`

*Returns*     <Nil> Dumps the specified objects to the file in Alpha\_1 text format.

*ObjectList*

      <listOf object | object> The objects to be dumped.

*FileName*    <string> Name of the file to put them in.

To dump a color attribute and a shell object for checking orientation as described in the previous section, we might use

```
dumpA1File( list( colorAttr, HandleShell ), "handle.a1" );
```

Data which is saved in this way can be loaded back into `shape_edit` using the `altose` program. This program converts the text file format into a Lisp source file which you can load into `shape_edit`. To use the program, you must load the commands using the "altose" keyword to `getcmds`, as in

```
getcmds altose
```

```
altose files
```

*Output*       textfile Rlisp source lines which will reconstruct the geometry stored in a file in Alpha\_1 text format.

*files*        <filelist> The files to convert.

There are two steps to getting the geometry back into `shape_edit` in this way. The `altose` program is first executed at the Unix shell level:

```
altose geom.a1 >geom.r
```

Then, in `shape_edit` the file is read in and evaluated with the `in` command:

```
in "geom.r";
```

Note though, that loading an Alpha\_1 text file back into `shape_edit` does not restore all the program state associated with the model. The information which is stored in Alpha\_1 text files is just the

geometry, not any construction information. So later modification of the model is not possible when it is loaded in this way.

Another way of saving the results of a **shape\_edit** session is with **saveModel**. This routine does save the program state associated with a model which was constructed in **shape\_edit**. For example, if you create a few points, use them to construct lines and arcs, use those arcs to create a profile curve and then create a surface of revolution with the profile, **saveModel** will save all the intermediate geometric constructions and their relationships. If the file is loaded again in another **shape\_edit** session and one of the original points is modified, then the entire model will be updated, just as it would have been in the original construction. Only the final geometry need be specified for **saveModel**. The routine will trace through dependency information to discover and save intermediate constructions which are relevant to the final geometry.

**saveModel( ObjStruct, File )**

**Returns** <Nil> Saves the model and any intermediate constructions to the named file.

**ObjStruct** <object | listOf object> The object(s) which are to be saved.

**File** <string> The name of the file into which the output will be written. The suffix ".se.r" is usually used for files saved in this manner.

**saveModelInWindow( Window, File )**

**Returns** <Nil> Saves all the geometry currently displayed in the given window.

**Window** <symbol> The name of the window containing the geometry.

**File** <string> The name of the file into which the output will be written. The suffix ".se.r" is usually used for files saved in this manner.

These files can be loaded back into **shape\_edit** with the **in** command, but a better way is to use **restore** which doesn't print everything that is happening to your screen. Depending on how the model was constructed, various parts of the model may have been named at the time it was saved. Although these names are saved, the **restore** operation does not normally bind these names back into the symbol table, due to the possibility of destroying data already in the **shape\_edit** that might have the same name. Instead all the objects are stored in a global vector called **ObjRefs!\***. It is a bit cumbersome to dig through that vector for the objects of interest, so several helper functions are provided. For display **showRestoredModel** will show all the objects which were read back in. If you really want to reference the objects by their names, you can have them put back into the symbol table using **bindModelToSymtab**. This should rarely (if ever) be necessary for working from the menu-driven interface, but will probably always be necessary in the command-driven interface. Once the names have been put back into the symbol table, **showNamedModel** will show all the restored objects which have names.

**restore( File )**

**Returns** <Nil> Reads the named file and reconstructs the geometry.

**File** <string> The name of the file which was saved using **saveModel**.

**showRestoredModel()**

**Returns** <Nil> Displays the results of the last call to **restore**.

**bindModelToSymtab()**

**Returns** <Nil> Puts named objects from the last call to **restore** back into the symbol table.

**showNamedModel()**

*Returns*    <Nil> Display just the named objects from the last call to **restore**.



## 13. Input for Utilities

The remainder of this manual primarily describes a number of utility programs which are used in conjunction with **shape\_edit**. These programs usually process data which has been constructed using **shape\_edit**. Most Alpha\_1 utility programs have an associated set of switches which control various parameters affecting the computations. The values of these switches are passed along when the program is executed, so they are set beforehand. All have defaults which are intended to be the most commonly used values.

Output from most programs goes to the Unix "standard output" or *stdout*. This is usually your terminal, and since many output forms are binary data, you will want to redirect *stdout* to a file. An example might be

```
render cup.al >cup.rle
```

The ">" symbol redirects *stdout* to the named file, in this case "cup.rle".

The input for most programs also comes from the Unix "standard input", or *stdin*, which can be redirected to come from a file with the "<" symbol. The commands which are used to execute Alpha\_1 utility programs, however, usually redirect *stdin* for you. So in the example above, the **render** command is actually redirecting the file "cup.al" to the standard input of the **render** program.

Aside from redirecting *stdin* for you, most of the commands for running Alpha\_1 utilities also concatenate a series of files if desired to form one input stream. The commands also convert from text to binary Alpha\_1 format (described further below). In order to provide all these functions, the Alpha\_1 command sets need to be loaded once per session as they are needed. The commands for each program are loaded with the **getcmds** command. Each program has a keyword (often the program name) which indicates to **getcmds** which command set is to be loaded.

```
getcmds programkey
```

Output     <Nil> Load a set of commands for a particular program.

programkey

<keyword> Indicator for which command set is to be loaded.

The Alpha\_1 system has two (equivalent) forms of data files. One is a human-readable text form, and the other is a binary form suitable only for programs to read. Almost all the Alpha\_1 programs deal exclusively with the binary form. As mentioned above, the commands for running utility programs will automatically convert your text file to binary form. The binary form of the data is considered temporary. Data should never be stored permanently in binary form, as the format may change without notice and the file will be unreadable with the revised **conv** command. For this reason, **shape\_edit** is an exception to the rule that programs always read and dump the binary form; **shape\_edit** dumps its data in text form.

If you need to convert a file from one form to the other, you must first make the conversion commands available. The keyword for the conversion command set is "conv". So you would say:

```
getcmds conv
```

To explicitly create a binary file, the **conv** program takes a stream of data (which may be binary, text, or mixed) and converts it to the binary form.

```
conv files
```

Output     <binarystream> Convert input text binary stream. The input may contain binary data mixed in with the text data.

files       <filelist> Files to be converted. They may be text or binary.

For example:

```
conv file1,file2,file3... >file.conv
```

Note that there are no spaces between the file names. Output is to standard out (usually the terminal), so it will be necessary to redirect the output to a file as shown above. (Typing a binary file to your screen may put your terminal in a bizarre unknown state.)

Since the output of most programs will also be binary, if the output needs to be examined, the `deconv` program will have to be used to turn the object stream back into a text form. For a detailed description of the text form, see [Text File Format], page 299.

`deconv files`

*Output*     <textstream> Convert input binary stream to pure text stream.

*files*       <filelist> Binary files to be converted.

The input data for `conv` (and hence all commands which use `conv` as a preprocess) comes from a directory which you may specify. The default the directory to which you are connected. The directory which is used may be reset by executing `setdatadir`.

`setdatadir dir`

*Output*     <Nil> Set the directory where programs will look for input files.

*dir*         <dirname> Directory from which programs are to retrieve the input files.  
If the directory name is missing, the default is reset to be the currently connected directory.





## 14. Combining Objects

### 14.1 Combiner Overview

The **combiner** is the mechanism which allows Alpha\_1 designers the freedom of modeling surface pieces which represent objects of interest without having to force the boundaries of parametric surface patches to meet, exactly, along surface "feature" boundaries. It implements the "boolean operations" which are the basic operations of the Constructive Solid Geometry (CSG) modelers which have been developed to allow modeling of a large class of objects from combinations of regular solid primitives. The extension of these boolean operations to sculptured surface primitives allows the designer more freedom in creating part models, but introduces complexity and subtlety to both the implementation and use of these operations.

The domain of discussion is broader when we discuss boolean operations in the context of a surface based modeler. The familiar operations of union, intersection and difference are intuitive and easily explained in terms of cubes and cylinders (solid objects) but are not as straightforward when applied to surfaces which are not "closed" and therefore do not trichotomize space into "inside", "outside" and "on" the part.

Therefore, we describe in more detail the shell object which was introduced earlier. A shell represents a grouping of surfaces which may be connected ("sewn together") along their appropriate edges by adjacency declarations. The boolean operations are stated in terms of these shells. The results of the boolean operations are represented by an algorithmic approximation of the intersection curves between these shells and regions of the surfaces bounded by these intersection curves. That is, the output of the combiner is a set of polygons and any surfaces which didn't have intersection curves.

The rest of this section provides an overview of the procedure for performing boolean operations.

One first describes the geometry of the surfaces which will represent the part to be modeled. This is covered in earlier chapters and it is assumed for this discussion that only a collection of surfaces is constructed (no procedures which generate shells have been used). Further, the surfaces have an inside and an outside, that is, the orientation of the surface normal is used to denote which side of the surface is the solid. Recall that the normals point "into the part".

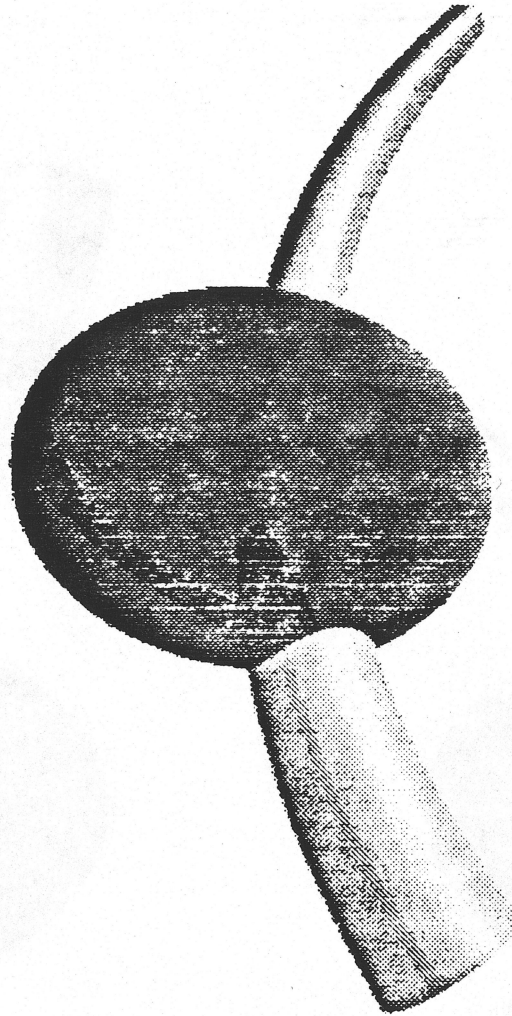
Second, construction of a shell from surfaces is done by:

- Declaring adjacencies between edges of surfaces which are shared.
- Grouping this collection of surfaces into a shell.

Third, the boolean operation which is to be performed is described in terms of a set expression which describes the desired combination of the component shells. This is a text expression with the operations of union (+), intersection (\*), difference (-) and negation (~). Figure 14-1 shows two objects rendered transparently, a "tusk" and a "grape" which are to be combined. Figure 14-2 shows the results of the set operations union, intersection, and both differences.

Conceptually, the model is completed. Operationally, the shells and set expression are written to files and the **combiner** program is invoked to produce a representation which can be used as the basis for analysis or rendering programs.





**Figure 14-1:** Tusk and Grape Example

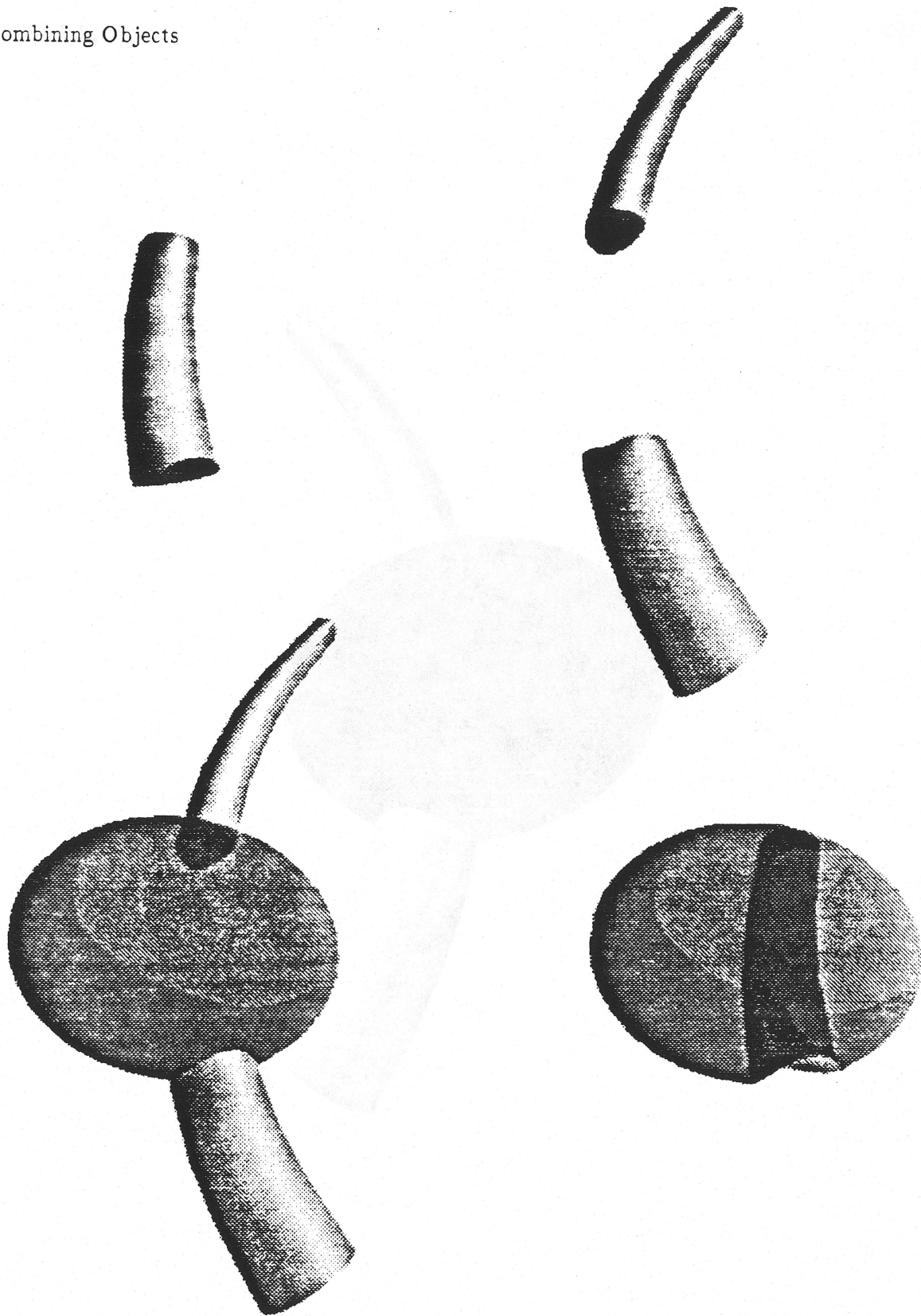


Figure 14-2: Boolean Operations on Tusk and Grape

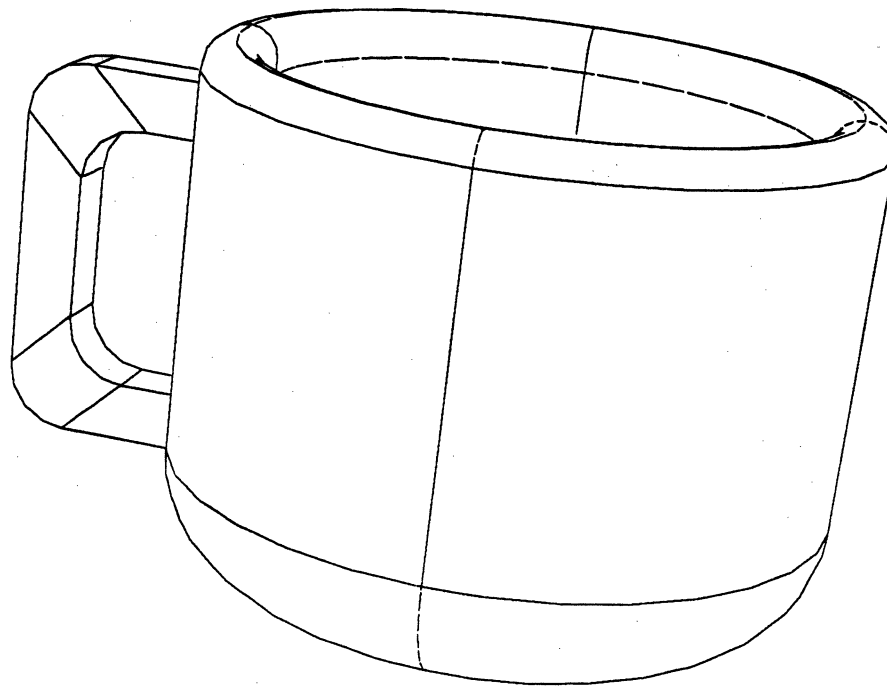


Figure 14-3: Example Coffee Cup

## 14.2 Preparing a Model for Combining

In order to discuss the preparation of the model, we consider a simple example object, a coffee cup. The cup consists of a body which has some aesthetic appeal, a handle which is a bent, tube-like surface, and two other surfaces, the floor of the inside of the cup and the base of the exterior of the cup. This is meant to be a topological description of the cup which allows many different geometries. One possible cup is shown in Figure 14-3.

This choice of example, using very simple geometry, i.e., a surface of revolution for the body, round planar surfaces for the floor and base and a simple tubular handle is meant to show a variety of ways in which surfaces are combined to make a solid. Specifically, the body will be attached to the floor and base using declared mutual adjacencies, the handle surface will be closed into a tube by using a self-referential adjacency declaration and the handle surface will be attached to the body using a boolean operation.

As one refined the design of the cup so that, for instance, the base of the cup became a more ornate pedestal or Grecian column, it might be more desirable to attach it to the body using a boolean operation. The strategy used to model an object may be easily changed as the design progresses. The point of this example is to illustrate different ways to join surfaces into solids.



### Preparing the Surfaces

The surfaces representing the object to be modeled can be constructed using a variety of basic construction methods described elsewhere in this manual. It is important to be aware of the surface normal convention which is used to distinguish the solid from the void during this construction. Surface normals point into the solid. You can check the orientation in `shape_edit` by setting the display of surfaces to include normal vectors at the four corners as described in the section on displaying objects (see section 6.3 [Displaying Objects], page 35).

The objects on which the boolean operations are performed are a higher level construct than surfaces for several reasons. These objects are called *shells* and they are composed in several ways. Historically, it has been common practice to model objects as collections of bounded parametric surfaces which meet at their boundaries. For example, a cylinder or a box can be modeled exactly, without need for any trimming or boolean operations by creating bounded surfaces which represent the "faces" of the object. This type of modeling provides a very concise means of representing shapes. In order to extend the utility of these surface models to the domain of solids which can then be combined using boolean operations, the connectivity of these surfaces must be made explicit so that the algorithm which evaluates the combination expression does not have to deduce information which is already known at an earlier stage in model construction. This information is used to connect intersection curve pieces which are later used to classify the surface segments of the resulting combined object.

This technique of defining surfaces which enclose a solid and declaring their connectivity can be used to encapsulate the definition of objects of interest which contain sculptured surfaces and thus are much more general than the simple primitives of CSG based geometric modelers.

It is also common in modeling to think of a geometric feature as a set of surfaces which are not contiguous, but which logically define a feature of the model, for example, if the cup design were extended to provide two handles, one still might think of the "pair of handles" as the handle when describing the whole cup as "body+handle".

In summary, a shell can represent

- a set of surfaces which enclose a solid,
- a set of surfaces which are contiguous, but don't enclose a volume, or
- a set of surfaces which are disjoint, but which logically represent a single geometric element.

### Declaring Surface Adjacencies

The basic surface modeling element of Alpha\_1 is the parametric, tensor product, B-spline surface. The characteristic which concerns us most when we are declaring adjacencies is that the surface is topologically rectangular, that is, when we look at it in the space of the parameters which define it, we can associate terms with the boundaries which uniquely identify each one.

Associated with the topologically rectangular surface is a control mesh which (partially) defines the shape of the surface. It is represented by a matrix of points. The boundary of the surface which is associated with the first row of the matrix is called the "top" boundary of the surface. The boundary of the surface which is associated with the first column of the matrix is called the "left" boundary. The "bottom" and "right" boundaries are the obvious mates. (Incidentally, as you look at the matrix in this way, the surface normal of the surface points into the solid which it represents. The normal is determined with a right-handed cross product  $U \times V$  at the top left corner.)

Given surface names and boundary identifiers, adjacencies are declared by the designer by indicating which boundaries are coincident. This specification is "loose" in the sense that the boundaries are not necessarily co-located over their full extent (i.e., they may only run together for short way)

and the adjacency may not necessarily be unique (i.e., a boundary may be adjacent to several other boundaries). Further, no indication of the parametric range over which they are coincident is required for this loose form of adjacency declaration.

Adjacencies are declared in **shape\_edit** with **mkAdjacent** (see the chapter **Leaving Shape\_edit**). There are two forms of adjacencies, *intersurface* and *intrasurface*.

If one surface was used to form a tubular handle for the cup, one could close the seam in the handle by the the following declaration:

```
mkAdjacent( HandleSrf, 'Left, HandleSrf, 'Right );
```

If one used a surface of revolution to form the inside and outside of the cup and several discs to make the inside floor and outside bottom of the cup, the body of the cup could be "sewn" together in the following style:

```
% Declare the attachment between the floor and the body of the cup
mkAdjacent( Body, 'Bottom, Floor, 'Right );
mkAdjacent( Body, 'Bottom, Floor, 'Left );
mkAdjacent( Body, 'Bottom, Floor, 'Bottom );
mkAdjacent( Body, 'Bottom, Floor, 'Top );
...
```

Fortunately, the code which generates shells of revolution does this so that the designer doesn't have to when using a higher level operation like **shellOfRevolution**. The **autoMkAdjacent** routine (see the chapter **Leaving Shape\_edit**) may also be useful for finding some adjacencies automatically.

### Building a Shell

A data structure which represents a shell is constructed from the surfaces which have been bound together. The name of the shell will be referenced later in the set expression which specifies combination with other shells. The use of the shell construction procedure is described in the section on shells (see section 12.1 [Shells], page 181). For the cup example, we might say:

```
Body := shell( list( Cup, Roof, Floor ), Nil );
```

The scope of the surface names used to declare adjacencies is limited to the same shell. That is, you can use the same name for several surfaces, as long as they are put in different shells.

### Boolean Combinations of Shells (Set Expressions)

An "algebraic" expression is used to denote how the shells which have been defined are combined to produce a desired object. The operators in the expression are "+" (union), "\*" (intersection), "-" (difference), and "~" (negation). Parentheses can be used to group the operations to make the sequence of combinations precise. Intersection is evaluated before union and difference, and negation applies only to the shell name or parenthesized expression immediately following.

The cup is a simple example of the union of two shells. The set expression is:

```
body+handle
```

The set expression can be included as a string attribute (at the front of the file) as in

```
CupExpression := attribute( "set_expr", "body+handle" );
```

or it can be supplied to the combiner later on the command line.

It is much better, however, to specify set expressions from **shape\_edit** using **combinerObject**. The *SetExpression* is an expression involving shells, parametric types, groups of shells, other combiner-Objects, ID's or strings, and the operations "+", "\*", and "-". (Use a unary "-" for set negation when creating combinerObjects in **shape\_edit**). A group of shells is interpreted as an implicit



union of the component shells. The geometry slot of any parametric types should evaluate to a shell. (This means that the **makeGeom** method procedure should build a shell).

**combinerObject( SetExpression )**

**Returns** <combinerObj> An object representing the specified boolean combination.

**SetExpression**

<expression> An algebraic expression involving shell names and the operators described above.

For example,

```
Comb := combinerObject( ( Shell1 + Shell2 ) * "shell3" );
```

Strings and ID's are allowed for referring by name to shells created in other **shape\_edit** sessions and stored in separate files.

### Saving the Data for Later Evaluation

The shells which are defined in the **shape\_edit** environment are written to a file so that they can be combined by a C program called **combine**.

This is done by **dumpA1File** which writes a list of "objects" (in this case shells) to a file. For the cup example:

```
dumpA1File( list( Body, Handle ), "coffeecup.a1" );
```

The ".a1" file extension is a convention which indicates that the file contains a text representation of the Alpha\_1 objects.

When a **combinerObject** is dumped, any attributes it possesses are dumped as global attributes. The set expression is written first, and all objects referred to by the expression (except strings or id's) are dumped as well. So, if you had created a **combinerObject** for the cup in **shape\_edit**, you could have just dumped that:

```
Cup := combinerObject( Body + Handle );
dumpA1File( Cup, "coffeecup.a1" );
```

## 14.3 Running the Combiner

Invocation of the combiner is similar to running other C utilities in that a set of commands are loaded to provide a simple interface to the underlying program. The keyword for loading the commands is "combine" (e.g., "getcmds combine").

The defaults which are in effect when the combiner is invoked are intended to accomodate normal usage and will be discussed in more detail below. The **combine** command expects a filename or a comma-separated list of files which contain the shells to be combined and a string argument which is the set expression which the combiner is to evaluate. The standard output is directed to a file which will get the results of the combination operation. Remember, this will be a binary file, so you don't want to print it on your screen.

**combine files expression**

**Output** <binarystream> Polygon and surface data representing the boolean result.

**files** <filelist> Files to be used in the boolean combination.

**expression** <string> Boolean expression describing the desired result. Usually this should be contained in the file. If so, just give the null string (quotes with

nothing in them) as the expression. Either single or double quotes are accepted.

For our example, the command is

```
combine coffeecup.a1 "body+handle" > cup.polys
```

This will start the combiner and, if all goes well, output to the terminal will list all the intersection curves found between surfaces, note that they were all connected into "loops" and provide some cryptic information about intersection points. The output file will contain a binary form of the polygons resulting from evaluation of the set expression which is suitable for viewing or rendering.

Note that when you use the **combine** command, if you want it to use a set expression in the file (as will be the case if you created a **combinerObject** in the **shape\_edit**) you must specify an empty set expression on the argument line, e.g.,

```
combine coffeecup.a1 '' > cup.polys
```

The **comb\_eps** command is used to set the intersection epsilon value. This determines the resolution in object space to which the surfaces will be subdivided before intersecting. The resulting approximations are guaranteed to be within the epsilon distance from the actual surfaces, but under some circumstances the generated intersection curve may not be this close to the real intersection. This will occur when the surfaces are almost parallel. The default setting to 0.05 works for parts which are four or five inches in maximum extent in English dimensions. If your part is Metric and small, use the **comb\_eps** command to set the epsilon to 0.1:

```
comb_eps epsilon
```

*Output*     <Nil> Sets the combiner epsilon.

*epsilon*     <float> Combiner epsilon. Smaller numbers give greater accuracy and more computation.

The **comb\_lines** and **comb\_nolines** commands turn the "lines" option on and off. This option of the combiner provides a means of getting just the intersection curves which are calculated early in the process so that you can see what they look like.

```
comb_lines
```

```
comb_nolines
```

*Output*     <Nil> Turn on (or off) the combiner lines option.

Repeating the previous **combine** command should produce the same summary message, but the program should exit gracefully leaving the output file full of intersection curves for your close examination.

The **comb\_state** command causes the current parameter settings to be displayed. The **comb\_normal** command resets the combiner state to normal default settings.

```
comb_state
```

*Output*     <text> List the current parameter settings which are different from the defaults.

```
comb_normal
```

*Output*     <Nil> Restore all the combiner options to their default settings.

There is one other flag for the combiner which is occasionally useful. There is no command for setting it, so it must be enabled on the command line when the combiner is invoked. This is the

“-c” flag, which allows specification of which shells are to be part of the output. This should never be necessary if the resulting object is a solid, but sometimes it's useful to be able to just trim pieces from a surface without producing a solid. For example, the slotted spoon construction in the tutorial section of this manual cuts holes out of a single surface (for simplicity in the explanation). In order prevent any parts of the surfaces which define the holes from appearing in the output, the combiner is run with the “-c” flag specifying which shells to output. The list of arguments for this flag can be separated by either commas or spaces.

```
combine slot-spoon.a1 '' >slot-spoon.polys -c spoonshell
```

## 14.4 Combiner Problems

If, by chance, things didn't go quite that smoothly, there are some common misunderstandings which may have caused the problem.

### Intersection curves not all “loops”

Sometimes the result announced on the terminal described more intersection curves than were expected and/or they were not all noted to be “loops”. This happens if things are not where you expected them to be or if a surface did not have the proper surface normal orientation to allow curve segments to be properly joined end to end.

The first thing to check is the information that the combiner printed out about loops. A loop is a closed intersection curve, and the combiner must identify every closed intersection curves as a loop in order for the boolean operation to succeed. For example, if you are trying to subtract a cylinder from some other closed object to make a hole through it, the combiner must detect two loops. If the messages do not indicate the expected number of loops, or if you are not sure what loops to look for try running the combiner again with the `comb_lines` option turned on. You can run the `view` program (see section 15.2 [Positioning For Rendering], page 206) to look at the intersection curve data in the output file. If the shape of the curves looks right, compare the number of things that look like loops with the number of loops the combiner reported. Failure of the combiner to close loops is almost always due to missing or incorrect adjacency declarations, although it may also be due to surfaces which are not correctly oriented. See section 12.3 [Surface Orientation], page 185 for some suggestions for detecting surfaces which are inside-out.

Verifying adjacencies is sometimes a tedious exercise for the designer. For a discussion of adjacencies and hints on getting them declared correctly, see section 12.2 [Attributes], page 181.

Other causes of unclosed loops are related to a basic misunderstanding of what boolean operations on open shells mean. The shells which you are trying to combine should look closed in the areas where they intersect so that you are thinking of operations on solids, not surfaces.

### Combiner finds many intersection curves

When the combiner finds many more curves than expected, one probable cause is that several shells are coincident or coplanar over a region instead of intersecting cleanly along a curve. The combiner is not able to handle regions of this type. If the edge of one surface lies in a second surface, the intersection between those surfaces may not be properly computed. Change the geometry. Coplanar surfaces do not define a single intersection curve. Possible error messages are “Internal intersection” or “Intersection ends in middle”.

**Intersection curves appear "ragged"**

Shells which meet in an "almost tangent" manner are sensitive to the approximation epsilon. A coarse epsilon may cause a very rough approximation of the curve and curves may not be easily formed into loops. Try increasing the `comb_eps` value, or changing the geometry so the intersection curve is more clearly defined.

**Sensitivity to certain symmetries**

The combiner forms a piecewise planar representation of surfaces in the process of approximating intersection curves and classifying surface pieces. It does this by recursively subdividing B-spline surfaces and eventually comparing the planar derived representations. Some instances of geometry are particularly incompatible with this approach, which relies on intersecting edges and planar pieces with some degree of numerical confidence. In particular, an example such as two concentric cylinders will always subdivide so that the edges of the planar approximations are aligned with each other, forcing "edge to edge" intersection to be done instead of the more tractable "edge to plane" calculation. This can be avoided in some cases, in particular the cases where objects have rotational symmetry, by rotating one of the objects so that its isoparametric structure is not aligned with the object which it will intersect.

This form of "tweaking" may be necessary to get the desired results.

**Surface degeneracies**

Surfaces which are modeled with degenerate edges cause problems. If an intersection curve passes close to a degenerate edge, it will most likely produce "Intersection ends in middle" or "Internal intersection" messages. Models with degenerate edges can be processed by combine, as long as the intersection curves do not come too close ("too close" depends on the value of the subdivision epsilon).

**Trimming problems**

Because the combiner works on a "trimming" paradigm, there must be something for it to "trim". One should allow "overlap" of shells on both sides of an intersection curve rather than attempting to make a shell boundary fall exactly on what the user hopes to be an intersection curve.

**Surface Classification Problems**

If all the intersection curves are loops, there are other causes of abnormal termination which are related to the next stage of combination, that is, surface classification. There are a number of problems caused by numerical round-off. These produce messages like "Internal intersection", "Intersection ends in middle", "point not on edge", and sometimes "X point follows an X point", where X is `START` or `END`.

A known frailty of the classification process is its inability to classify surface pieces in the neighborhood of "extraneous intersection curves".

A more elaborate example is necessary to describe this problem. Consider two hollow, spherical objects. Each can be modeled as an "outside" sphere minus an "inside" sphere of smaller diameter. Now consider the object formed by placing these objects so that they interpenetrate and taking their union. The "feature" curves of the resulting solid object are a circle where the exterior spheres intersect and a circle where the interior spheres intersect.

There are other shell-shell intersection curves produced during the combination process, specifically the two intersection curves between the exterior shell of one object and the interior shell of the other. These do not contribute to the resulting geometry and are hence "extraneous". Looking at them in relation to a single surface, they form "loops within loops".

To avoid having the combiner get fatally confused by this situation, human intervention

is required. This help is provided by the user declaring surfaces which intersect to form these extraneous intersection curves "disjoint" so that the intersection is not attempted by the combiner and the intersection curve does not exist at the classification stage.

Each shell can be given a 'DISJOINT' attribute which declares it disjoint from the other shell e.g.,

```
setAttr( ShellAoutside, 'DISJOINT, "ShellBinside" );  
setAttr( ShellBinside, 'DISJOINT, "ShellAoutside" );
```

## 14.5 Summary of Hints

This section contains a very short summary of suggestions for using the **combine** program. All of the items are described more fully in the main body of this chapter, but this list can be helpful as a checklist before you run the combiner and if you run into trouble. You should think about the first four of these every time you create new geometry for the combiner. The others may or may not be relevant, depending on your model.

- Objects to be combined must have clear intersection curves. Extend them a little past each other. There must be no coplanar faces. Even faces that just barely touch each other can cause problems.
- Make sure all the surfaces are oriented correctly.
- Make sure adjacencies have been declared wherever surfaces meet each other.
- Choose an appropriate epsilon for the size of your part.
- Use the **comb\_lines** option to check that the intersection curves are what you think they should be.
- Watch for symmetries. Rotate cylinders slightly about their axes if they are going through the exact center of another primitive.
- Watch for intersection curves that pass close to degenerate edges of a surface.
- Check for surfaces that should be declared "disjoint".



## 15. Rendering Preparation

This chapter describes the various utilities available for previewing and selecting parameters for producing high-quality raster images. It provides suggestions on image composition, lighting, and color selection as well as describing several programs which aid in setting up parameters for producing good images.

### 15.1 Rendering Guidelines

This section discusses various aspects of producing shaded raster images of objects which have presumably been designed using **shape\_edit**. Producing good pictures (especially for photographing to provide hardcopy documentation of modeled parts) is a much more difficult task than is generally recognized, partly because the time it takes to generate a shaded raster image makes it difficult to experiment with minor adjustments to rendering parameters. The sections below describe the issues that must be addressed to generate high quality shaded raster images as well as a number of utilities available in the Alpha\_1 system for selecting parameters that affect the images. The actual rendering program itself is described in the next chapter. This chapter addresses the higher level issues which must be considered before the rendering program is invoked.

#### Image Previewing and Composition

The first major issues in generating an image include checking that the model has been constructed correctly, deciding what angle the object is to be viewed from in the final image, and deciding where it will appear on the screen.

One way to generate viewing transformations is with the **viewMat** function in **shape\_edit**. The resulting matrix can simply be dumped to the text file with the other geometry.

Another major utility for previewing and positioning objects is the **view** program which is described completely in a later section of this chapter (see section 15.2 [Positioning For Rendering], page 206). The **view** program displays objects on a line drawing display which is equipped with physical knob devices or a tablet which can be used to dynamically adjust the view. The knobs are used for manipulating the object to the desired orientation and position on the display. A transformation matrix which describes the resulting position may be output from the **view** program and used with the object data as input for rendering. It is now possible to generate this transformation matrix from **shape\_edit** on some devices (if a "send matrix" or "write matrix" command is included in the window manager functions).

When positioning your objects in the **view** program, fill the frame to get maximum spatial resolution out of the display medium. Center the subject matter and compose for even distribution throughout the picture area. However, if you want to make hard copy photographs, don't make it too tight at the edges; leave a little room for the photolab.

Once the objects to be rendered are positioned using **view**, there are several "fast" output modes of the rendering program which can provide valuable information without the expense of computing a high-quality rendering. *Flat shading* can be set so that each polygon which is derived from the surfaces for rendering purposes is shaded a single uniform color. This produces a faceted image, rather than a smooth one, but it may be useful for checking various parameters. *Smooth shading* can be used for a better image than flat shading, but not as good as the default *normal interpolation shading*.

The **resolution** attribute of the surfaces being rendered may also be adjusted so that fewer polygons are generated to approximate the surface. The default resolution is 1.0, which is generally as low as

the resolution should ever be set. It produces high-quality images which show little or no evidence of the underlying polygonalization in the rendering. Resolution values up to 20.0 and occasionally higher will still provide images where basic geometry can be verified, although values between 5.0 and 10.0 are more usual.

Another way to speed up the rendering process for quick previews is to eliminate all polygons in the resulting surface approximation which do not face the view. (The rendering flag which controls this is set with the **cull** command.) For objects which are closed, culling the backfacing polygons will not affect the image at all (unless the object is inside out, in which case you will see only the part this is supposed to be invisible!).

Generally, raster images are not filtered until a satisfactory unfiltered image has been successfully rendered. A filtered image takes much longer to generate than a similar unfiltered image. The unfiltered image may contain very noticeable "jaggies" or "stair-steps" along the edges of the objects which are a discretization effect. Filtering in the final image will greatly reduce these edge effects.

Finally, some additional utilities may be used for previewing images if special displays are available. The **render** program converts polygons to "pixels" on the screen in software, but there are display devices which accept polygons as primitives. One of these is the Lexidata SolidView. Two programs which are used to quickly display surfaces on the Lexidata are described in the section on fast polygon programs (see section 17.3 [Fast Polygon Programs], page 233).

### Color Selection for Images

Selecting colors for the objects to be displayed and the background is the next step in preparing to render an image. Colors must be specified in RGB (red-green-blue) components. Since this is usually not an intuitive color blending space (red and green together make yellow, for instance), the **colors** program (at Utah only) may be used to select colors by adjusting RGB or HSV (hue, saturation, value) interactively. The RGB coordinates for a chosen color can be printed from the **colors** program. Once the RGB coordinates are selected, the color can be assigned to objects in **shape\_edit** by setting color attributes.

If any of the objects are to be transparent, the color selection process is even more complex. The **tcors** program (also only at Utah) may help in selecting colors that will interact well for particular transparency values.

As a general guideline for selecting colors, try to make high-contrast pictures rather than dull, flat ones. If you sit in a dark room and look at a monitor, your perception of the picture is quite different from a print of the same image. Pictures are usually too dark, so use the high end of the intensity spectrum. Images which contain pairs of colors which are complementary have more visual impact.

Background colors are as important as the choice of colors of the objects in the image. White backgrounds are generally not good, and are very bad for photographing. It is hard to make a good, really white print. A black background or deep blue one tends to maximize the visual impact of the subject matter. Red or Yellow backgrounds bleed onto the subject. You could use light magenta or pink as a background, but a dominant background changes the way one visually perceives the subject matter.

### Light Vector Selection for Images

Finally, the choice of positions of the light sources has a great deal of importance in the quality of the final image. Generally, several light sources will be necessary, positioned so as to highlight the most important features of the objects in the image.

The overall effect of the light vectors is partially determined by certain shading parameters and



material types which are described in the Shading Parameters section (see section 16.2 [Shading Parameters], page 222).

A very useful program for selection of light vectors is the **shade** program (again, available only at Utah) which is described in detail in the section on selecting shading parameters (see section 15.3 [Selecting Shading Parameters], page 207) later in this chapter. An image must be rendered using the **dynamic** option of **render** and displayed on the frame buffer. The **dynamic** option builds a buffer of normal vector values rather than RGB intensities. The **shade** program then uses this information to allow interactive positioning of light sources, adjustment of shading parameters and some selection of color values. Lights can be selected in a number of ways, including picking a particular point on the image which should contain a highlight and allowing the program to compute an appropriate light vector based on the normal information.

### Display Selection

The final decisions to be made in preparing to render an image involve the display which will be used. Different displays have different aspect ratios. This means that the individual picture elements (pixels) of the display may not be square. Filling in a 100-by-100 box on such a display may result in a box which does not have the same width as length. For accurate images of geometric objects (especially those with certain kinds of symmetries) the aspect ratio of the display device must be calculated and compensated for in the rendering program. **Render** compensates by stretching the picture by an appropriate amount in either the X or Y direction. (The aspect ratio can also be adjusted with the **fant** program, which is part of the Utah Raster Toolkit and distributed with Alpha\_1.)

**Render** never draws directly to a display, but rather generates a run-length-encoded (RLE) file which describes the image produced. The **buffer** option to **render** specifies the name of the file to which the RLE information is written. The RLE file may then be displayed on any of a number of raster devices which can interpret the RLE format. However, some devices have a lower resolution than others. An image which is generated for a 1024x1024 resolution display cannot be shown on a 512x512 resolution display unless some compression or filtering is done. The 512 image can be shown on the 1024 display, but will only fill the lower left quarter of the screen unless some expansion is done. So you will have to decide on the primary device which the image will be displayed on, and **render** to this resolution by setting the appropriate **render** options.

### Image Debugging

One of the most common problems in rendering an image is that the shading is not as interesting as expected. This may be due to the surfaces being inside out. Try the **flip** option to **render** to invert normals during rendering, or specify a **back\_color** for the objects that is much different than the normal color. If the surfaces are wrong, they should be corrected in the modeling of the objects (the **shape\_edit** procedures). Although pictures can be generated with the **flip** option, boolean combination operations will not succeed on surfaces that are incorrectly oriented.

Sometimes it may be useful to see how the **render** program approximated the spline surface as polygons. The **quilt** option to **render** ignores the specified surface colors and assigns neighboring polygons different colors so that the polygonal approximation becomes visible.

The **mung** program is also sometimes useful for adjusting parameters such as color, transparency, and resolution in a large text file (e.g., the output of the boolean combination program) without regenerating the model to adjust minor attributes. **Mung** is described in more detail in the section on modifying text files (see section 15.4 [Modifying Text Files], page 212) later in this chapter.

## 15.2 Positioning For Rendering

The **view** program displays an Alpha\_1 model on any of several line-drawing displays. It allows real-time translation, rotation, and scaling of the model for visualization, and to prepare transformation matrices which position objects on the screen for raster rendering via the **render** program. It also provides a quick way to examine the results of a boolean combination operation.

The keyword for loading the view commands is "view" (so you can say "getcmds view").

**view files**

**Output**     <nil> Display geometry stored in files on the currently selected graphics device.

**files**       <filelist> The files to view.

After orienting the object, you may write out the transformation matrix. Suppose you choose to write it to file "a.mat" on your connected directory. You can then get the same orientation in your rendered image by saying

```
render a.mat,file1,file2...
```

On some devices, **view** automatically sends the output to a file named "a.mat", while on other devices you may specify the filename. In the former case, it is usually a good idea to copy a.mat to another filename if you wish to use it again because otherwise a.mat will be overwritten the next time you start up **view**. (Note that the a.mat file is cleared at startup, whether or not you intend to write a matrix.)

The following set of flags can be used to control the appearance of curves and surfaces in view and other programs that display splines:

**smoothcrvs**

**nosmoothervs**

      If set, draw curves smoothly (default on).

**iso**

**noiso**       If set, draw isoparametric lines on surfaces (default no).

**polys**

**nopolys**    If set, don't draw curve control polygons (default off).

**meshes**

**nomeshes**   If set, don't draw surface control meshes (default off).

**defsrffineness**

**srffineness value**

      <float> Use default isoparametric line spacing (default 50).

**defcrvfineness**

      Use default curve fineness (1).

**crvfineness value**

      <float> Set curve fineness.

**center**

**nocenter**   If set, the object is centered on the screen to start (default on).

Default display is an X device. The setup commands listed below change the target display to MPS, PS300, Iris, or X device, respectively.

**viewps**

**view300**

**viewiris**  
**viewX**

The PS300 version of view works slightly differently from the other versions, because the PS300 is capable of running independently from the host. Since it runs independently, you must use the **get300mat** command to enable the function keys that write the transformations matrices out. The **get300mat** program waits for matrices to go into "a.mat" when the "write mat" button is pressed, and exits when the "quit" button is pressed. This can be done at any time after the initial view command is started, as long as your view is still on the PS300 screen. Alternately, the **view300mat** setup command causes view to automatically do a **get300mat** at the same time.

**get300mat**  
**view300mat**

On X devices, use the "more" menu button to get to the commands for writing out the transformation matrix.

### 15.3 Selecting Shading Parameters

The **shade** program (which is only available at Utah in general) allows interactive adjustment of lighting directions and shading parameters. The program uses both the MPS and the Grinnell frame buffer. Prior to running **shade**, the image to be studied must have been rendered and displayed on the frame buffer using the special **dynamic** mode which stores normal direction codes rather than the computed color values. Since **shade** modifies the image in the frame buffer, it is strongly recommended that the image be saved **before** running **shade** (or by selecting the "Save Image" menu button in **shade**).

The **shade** program includes features for:

- interactive positioning of multiple light sources
- selection of a color to be used in displaying the image in "psuedo-color" mode
- adjustment of the various parameters for Blinn and Cornell shading
- a second light source opposite each primary one
- flipping the orientation of the displayed surfaces
- display of the current shading function as a 3-dimensional graph of normal direction versus intensity.

#### Using Shade

The **shade** commands are loaded with **getcmds** using the keyword "disp" (as in "getcmds disp").

To use **shade**, first load the specially computed image into the frame buffer. If the image was produced by **render** with the **dynamic** option, then it must first be pre-processed. If it was saved using the "Save Image" menu button of **shade**, then it is ready to go. Note that "Save Image" puts information about surface normals into the color map, so care must be taken not to destroy the color map information before running **shade**.

To start the **shade** program, use the **shade** command if there are no input files and the **cshade** command if there are input files.

**shade**

*Output*      <textfile> Shade writes out the set of parameters which are active when the program exits.

**cshade files**

**Output**      <textfile> Cshade writes out the set of parameters which are active when the program exits.

**files**          <filelist> Parameters to be used to initialize the program.

The input for **cshade** is standard Alpha\_1 data files which may contain colors, light vectors, blinn\_params and dist\_params. If present, these are used to initialize the **shade** program parameters. The output of **shade** or **cshade**, when you "quit" normally, is the currently selected set of parameters, so you must be sure to specify an output file if you want to use that data later for rendering.

Flags may be set for **shade** using the following switches (defaults are given first):

**calc\_norms**

**nocalc\_norms**

If set, calculate the normal discretization. This flag must be specified if the image was created by the **dynamic** option of render. (default off).

**shadedbg**

**noshadedbg**

If set, displays a neat picture of what it's doing during calculation of normals. Ignored if **calc\_norms** is not also specified. (default off).

**background red green blue:**

**nobackground**

If set, use the given background color. (default black background).

The **shade** menu may seem complex at first, but it can be broken down into a few basic regions. It occupies an "L" shaped region on the screen, as shown in Figure 15-1.

The regions of the screen have the following functions:

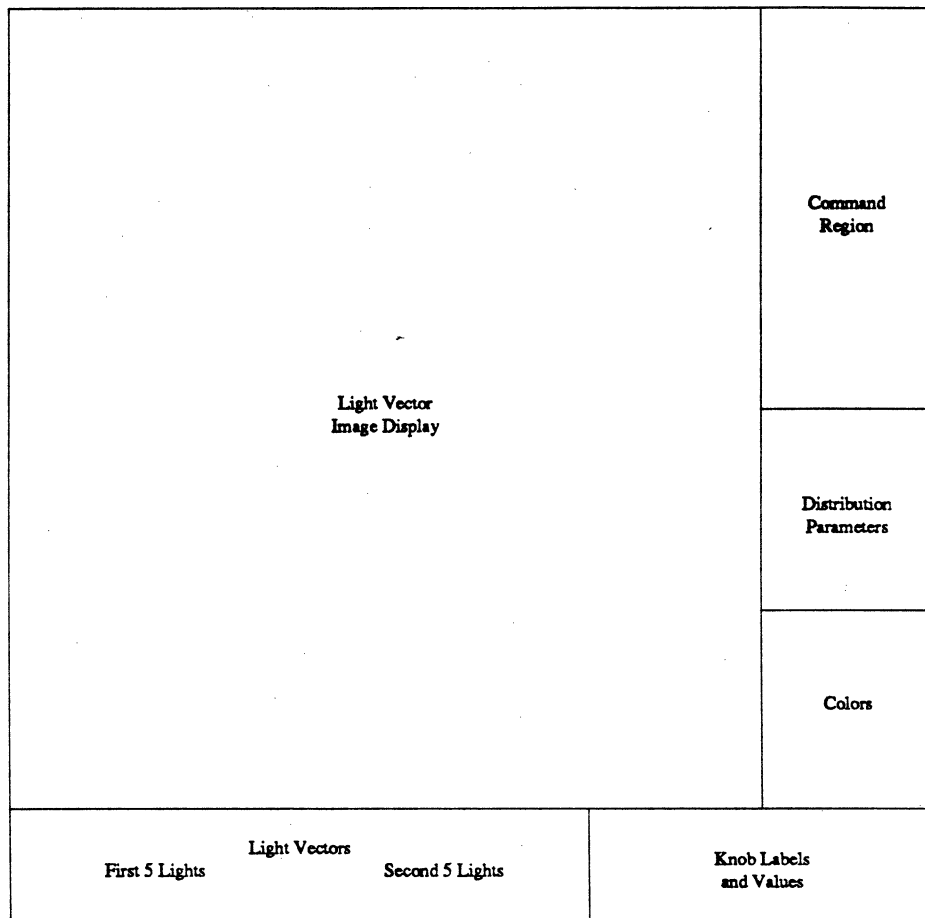
- The square *Light Vector Image Display* region in the upper left shows a three-dimensional view of the light vector and a plot of the intensity of the reflection for each surface normal.
- The *Command Region* contains most of the menu commands.
- The *Dist Params* area shows up to 5 distribution parameters with an arrow ">" marking the one which may be modified using the knobs.
- The *Colors* area shows up to 5 colors with an arrow marking the one currently displayed on the frame buffer screen.
- Up to 10 light vectors are displayed numerically in the *Light Vectors* area of the screen. The currently selected light vector is marked with an arrow. Only this light vector is shown in the *Image Display* area.
- Finally, knob labels are displayed in the lower right corner, together with their respective values.

The individual menu commands are explained below.

**Quit**          This button is picked to exit **shade**. It must be picked twice in a row. The first time, the word "Quit" is uppercased to indicate the pick. If any other menu item is picked, the highlight will be removed, and **shade** will not exit.

**Save Image**

Pick this button to save the discretized normals using **svfb**. A file name is prompted



**Figure 15-1:** Layout of Display for Shade Program

for, and **svfb** invoked to save the image. The surface normal information is saved in the color map.

#### **Display Normals**

The display of the 3-D normals plot sometimes takes a significant amount of time. Therefore, this button is supplied to suppress the display.

#### **One Light/Two Light**

This button toggles between two states. In the **One Light** state, each light has the expected effect. In the **Two Light** state a second, diffuse light source is placed diametrically opposite each light. This often gives better looking shading when a single light is used.

#### **No Flip/Flip**

This toggle is used to compensate for inside-out models. When in the **Flip** state, all

surface normals are negated before computing the shading function.

**Add Light** Picking this button adds another light source. All lights (except the first) are initially at (0, 0, 1). Note that there is no way, currently, to delete a light source.

**Front Light/Back Light**

This button works together with the tablet positioning of the light vector. Any given (x,y) position of the light corresponds to two z values, one positive and one negative. If this button reads "Front Light", the z component will be positive, if "Back Light", the z component will be negative.

**Cosine/Blinn/Cornell**

This button cycles among three states, starting at **Cornell**. It selects the shading function being used. Both **Blinn** and **Cornell** shading work with multiple light sources, but **Cosine** shading ignores all but the first.

**Log Map/Linear Map**

The **Log Map** selection causes the shading function to be mapped through a color compensation function before being loaded into the frame buffer. Since we generally use this same compensation table as our color map, this button will normally be left in the **Log Map** state.

**Metallic/Plastic**

This toggle button selects either metallic or plastic looking highlights. (Metallic highlights are the color of the material, while plastic highlights are always white.) Even when the final image will have metallic highlights, it is often easier to position the highlights when **Plastic** is selected.

**Set Hilite Posn**

When this button is picked, a cursor is displayed on the frame buffer. This should be positioned over the point on the image at which a highlight is desired, and the tablet button (or "pen") depressed. The light position necessary to place a highlight at the indicated point will be calculated, and the currently selected light vector will be moved to that position. If a "focus of attention" is selected (see below), it will be moved to the indicated screen position as well.

**Focus on Center/Focus [x, y]**

The **good\_highlights** option to **render** causes the direction to the eye to vary depending on the position on the screen, correctly modeling a finite eye position, as required by the perspective transform. It is not possible to do this correctly for all screen points with the dynamic shading process; the same eye direction must be used for all points on the screen. With **Focus on Center** selected, the image displayed appears as it would with **good\_highlights** disabled (the default state for **render**). To get an idea of what it would look like with **good\_highlights** turned on, the apparent eye vector may be moved so it is correct for any given point on the screen (and incorrect for all others) using **Set Focus** (below). When **Focus [x, y]** is selected, x and y indicate the point on the screen for which the shading is correct.

**Set Focus** Selecting this button causes the cursor to appear on the frame buffer. A point in the image may then be selected, and the shading will be recalculated with the eye vector appropriate to the selected point. Note: **shade** assumes the standard, default perspective transformation matrix in this calculation. Selecting a highlight position (above) will also shift the focus of attention to the highlighted point.

**Add Dist Param**

Selecting this button will add a distribution parameter pair to the list. Up to five distribution parameter pairs may be active. The values for the currently selected

distribution parameter pair can be modified with the knobs. No attempt is made by shade to ensure that the sum of the weights of the distribution parameters is anything reasonable.

#### Distribution

Selecting any of the active distribution parameter pairs will make it the current one.

#### Add Color

Selecting this button will add another color to the list of available colors, up to a total of five colors. Only one color can be displayed at a time, however. The initial value for each color is selected from a predefined list. The current color may be changed by picking the desired color. If the current color is picked, a color modification menu is displayed on the frame buffer. While the (frame buffer) cursor is in the color selection area, a "normal" color map is loaded, and the displayed image will not be correct. If the cursor is moved into the image area, the shading function will be loaded, using the new color. To exit color selection, simply press the button while the cursor is in the image area.

#### Light Vectors

The current light vector may be selected by picking it.

#### Image Area

The light vector may be manipulated with the tablet by picking points in this area. The end of the light vector will be positioned at the indicated (x,y) position, with the z coordinate either positive or negative, as indicated by the **Front Light/Back Light** menu button. [Note: on the MPS, the vector is currently somewhat shorter than the indicated position.]

#### Theory of Operation

The **dynamic** option to **render** causes the surface normal at each point to be mapped into the "color" of the image at that point. The normal is quantized into 8 bits each of X, Y, and Z. In this form, the image cannot be used directly by **shade**, but must be first preprocessed to reduce the total number of represented normals to be less than the number of entries in the frame buffer color map. **Shade** will do this if the **calc\_norms** option is specified. This second quantization procedure is done adaptively, and is generally different for each image. An attempt is made to quantize finely where many pixels have normals "near" each other, and coarsely when only a few pixels have a given normal value. Thus, each normal in the original image is assigned an *id*, and each *id* has a *representative normal* which will be used to compute the shade assigned to that *id*. At this point, the image may be saved, with the representative normal for a given *id* stored in that slot in the color map. If **calc\_norms** is not specified, **shade** expects the frame buffer to contain such an image and color map. The mapping of many normals onto a single *id* accounts for the faceted appearance of a dynamically shaded image.

After quantization, the dynamic shading process is conceptually simple. When any shading parameter (such as the light vector) is modified, a shade is calculated for each representative normal, the current color is multiplied by this shade and loaded into the correct slot in the color map. Thus, pixels containing the *id* corresponding to this normal will have the correct shade. Since there are typically 255 (or 4095) *ids*, this calculation is much quicker than recalculating the shade at each of 1/4 million (or 1 million) pixels, and can be accomplished in a few seconds.

When the shade for a given normal is computed, the program has no knowledge of where that normal might appear on the screen, and thus cannot vary the eye vector according to the screen position. Therefore, the entire image must be calculated with a single eye vector, and the **good\_highlights** option of **render** cannot be simulated correctly. It is possible to get the calculation correct for any given screen position, however, by using the eye vector as it would appear at that point. The

approximation is good near the selected point, and is worse for points further away.

Highlights are affected by the apparent eye vector much more strongly than is the diffuse shading component. Therefore, if a focus position is selected, positioning a highlight will move the focus of attention to that point, on the assumption that the image will eventually be rendered with `good_highlights` turned on. A little experimentation will show that moving the focus of attention from one side of the screen to the other changes the positions of highlights markedly.

#### Known Bugs

Currently, there is no way to delete an unwanted light vector or distribution parameter pair. The effect of a distribution parameter may be negated by setting the weight to zero, but it will still slow down shading calculation.

The MPS knobs must be turned slowly, always giving the program time to "catch up". If turned for more than 1/2 revolution between samples, the effect will be that of turning the knob in the other direction.

The light vector positioning by tablet is not quite correct on the MPS. This is due to interaction with the viewport used to display the light vector, and can be corrected fairly easily.

## 15.4 Modifying Text Files

The `mung` program is used for changing certain parameters throughout a data file, whether text or binary. It is most useful for adjusting rendering parameters of large data files which may be expensive to regenerate (e.g., the results of combining objects).

The `mung` program currently will change opacity, rgb, or resolution values in the following objects:

- opacity
- rgb
- resolution
- srf\_object
- polygon
- line

Command line arguments specify which values are to be changed and what the new values are. Every occurrence of the value(s) of interest is changed to the new value.

The commands for running `mung` are loaded with the render commands, so use that "render" keyword, as in "`getcmds render`".

```
mung [ -t opaq ] [ -c red green blue ] [ -f resolution ] [ -q ] [ -x xmin ] [ -X xmax ] [ -y
ymin ] [ -Y ymax ] [ -z zmin ] [ -Z zmax ]
```

**Output**      <binaryfile> Attributes of objects in the input stream are adjusted as indicated by the flags.

**-t opaq**      <float> New opacity value to replace any existing ones.

**-c red green blue**  
                <integer:0-255> New color value to replace any existing ones.

**-f resolution**  
                <float> New resolution (for rendering) to replace any existing ones.

**-q**            <Nil> Sets the quilt flag, which applies a semi-random coloring to polygons in the input stream, so that the polygonal approximation to a surface may



be visible in an image. It is similar (but not identical) to the effect of the **quilt** flag in **render**.

**-x xmin -X xmax**

**<float>** Set the x coordinates of a bounding box for culling objects that are not in the boundaries. Any object which does not lie within the given boundaries is deleted from the object stream.

**-y ymin -Y ymax**

**<float>** Similar y coordinate bounds.

**-z zmin -Z zmax**

**<float>** Similar z coordinate bounds.

As an example, the following command would change every occurrence of a color parameter in the input stream to gray:

```
mung file1.a1,file2.a1 -c 100 100 100 > filegray.bin
```



## 16. Shaded Rendering

This chapter describes the **Alpha\_1 render** program which produces shaded images for frame buffers. **Render** operates with a traditional scanline polygon fill algorithm and a Z-buffer hidden surface algorithm.

Images are rendered into a buffer file or standard output using a device independent run-length-encoded (RLE) format. The **getfb** program can then be used to display RLE files on the Grinnell frame buffer. Other RLE interpreters are available for other devices: **getX** for the X window system, **getc3d** for the Chromatics, **getbob** for the HP bobcats, and **getiris** or **getmex** for the Iris.

**Render** is a large program with many options available to the user. There are two types of controls: switches and parameters. Switches are passed to the program at run time as part of the command execution. Parameters, on the other hand, are part of the input data to the program. For example: the type of lighting model used to do the shading is chosen by a **render** program switch, but the number of light sources and their respective positions are parameters, or part of the input data. Switches and parameters will be described in separate sections although they must interact and be consistent for correct results.

The parameters that we will be concerned with are the ones that affect the shading of the image. These parameters include the color, transparency, and material properties of the surfaces rendered as well as the positions of the light sources. Each parameter can be specified by the user by including that parameter, using the required syntax, in an input text file. See [Text File Format], page 299 for detailed information on the formats.

The following sections describe the various switches and shading parameters in detail including their meaning, specification, and default values. Examples are provided on how to run the **render** program and use the flags and parameters. Lastly, two sections are provided for quick reference. One gives a brief description of the various **render** switches, and the other describes the syntax for specifying the various shading parameters.

### 16.1 Running Render

#### Choosing Render Options

The commands that control the **render** program are loaded with **getcmds** using the "render" keyword, as in "getcmds render". This defines all the necessary commands for setting and checking the various switches of the **render** program.

There are many switches, and a default value is provided for each one. To see what the default values are, use the **show\_defaults** command.

**show\_defaults**

**Output**     <textfile> Description of the default settings of render switches.

To see the options that are currently different than the defaults, use the **state** or **allstate** commands. The **defaults** command is used to reset all the options.

**state**

**Output**     <textfile> List values of commonly used render switches which are different from the defaults.

**allstate**

**Output**     **<textfile>** List values of all render switches which are different from the defaults.

#### **defaults**

**Output**     **<Nil>** Reset all the render switches to their default settings.

Individual switches can each be set by their own command. The section on render options describes these switches in detail, see section 16.1.1 [Render Options], page 216. A discussion on using shell scripts for rendering is also included, see section 16.1.2 [Scripts For Rendering], page 221. These scripts help keep track of the rendering process and make it possible to regenerate images long after the initial rendering.

### **Running the Render Program**

To render an image to an RLE buffer file simply give the following commands, one could simply say

```
render file1,file2,file3 >out.rle &
```

Any options set by the user before invoking this command are automatically passed down as arguments to the program.

#### **render files**

**Output**     **<rlefile>** A run-length encoded (RLE) image of the objects in the input stream.

**files**       **<objectstream>** The geometric data which is to be rendered.

The **render** program tends to be quite slow for an image of any complexity at all. If **render** is started as a background process (as indicated by the "&" in the example above), its progress can be checked periodically by running **getfb** or one of the related RLE decoding programs even before the image is finished. This will not interfere with the successful completion of the picture.

## **16.1.1 Render Options**

This section describes the meaning and use of the various switches for the **render** program. The options are divided into three major categories: switches that affect the shading, switches the control more general graphics utilities, and switches that affect the form of the output.

There are several commands for controlling each switch. Usually there is a command to turn a switch on and one to turn it off. Some switches have values other than "on" and "off". These usually have a command for setting the value and a command for restoring the default value. The default appears first in each group. Finally, it is noted that certain switches require other switches to be set. These interactions between switches are usually done automatically by the commands which set them and will be noted in the descriptions.

### **Shading Switches for Render**

There are three reflectance models to choose from (see section 16.2 [Shading Parameters], page 222 for descriptions of these). Executing one of these commands chooses the appropriate model.

<b>cornell</b>	Cornell lighting model.
<b>blinn</b>	Blinn lighting model.
<b>cosine</b>	Cosine lighting model.

The next group of switches chooses the type of interpolation that is used across each polygon rendered. **Normal** chooses a linear interpolation of the polygon normal vectors. **Flat** chooses flat shading of each polygon. **Smooth** uses shade interpolation to color the polygon. The **dynamic** command sets up an image to be used with the **shade** program (see section 15.3 [Selecting Shading Parameters], page 207), writing normals to the output file rather than RGB values. **Pseudocolor** mode uses color interpolation across polygons.

**normal**      Use normal interpolation.  
**flat**        No shade interpolation across polygons.  
**smooth**      Shade interpolation across polygons.  
**dynamic**     Set up for using dynamic shading program.  
**nopseudocolor**  
               No color interpolation (use normal shading).  
**pseudocolor**  
               Color interpolation across polygons with no shading.

Polygon normals can be flipped to the reverse orientation.

**noflip**      Leave mesh orientations as they are.  
**flip**        Reverse the orientation of all surface meshes.

The viewer (eye) is usually assumed to be at infinity which means that the vector from the object to the viewer is constant. This works in most cases; however when objects are viewed nearly edge-on problems can occur. The **goodhighlights** flag forces a vector to be computed from every pixel to a viewer at a finite distance. This gives better results but causes a considerable slowdown in computing the image.

**fasthighlights**  
               Viewer at infinity.  
**goodhighlights**  
               Viewer at finite distance.

The highlights of the image can be colored in two ways: one is the color of the object, and the other is to color the highlights white. The **white** and **nowhite** commands control this option. The **white** flag has no effect unless **normal** is set and either Blinn or Cornell shading is chosen.

**nowhite**     Highlights same color as object.  
**white**        White highlights (blinn/cornell only).

The light sources, which are input parameters (see section 16.2 [Shading Parameters], page 222), can be interpreted as single sources of light in the given directions, or as double sources with light coming from two opposite directions. The double light sources, while not as realistic, are useful for getting the maximum amount of information from a single image, since no areas will face completely away from both lights.

**twolight**    Double light source.  
**onelight**    Single light source.

Objects can be transparently rendered using the **transp** command. Transparent rendering also requires the use of normal interpolation, which is also selected by the **transp** command. Also note that opacity values must be set for each surface in the data file (see section 16.2 [Shading Parameters], page 222).

**notransp** All objects rendered opaque.  
**transp** Turn on transparent rendering and normal interpolation, only useful if the objects have transparency values.

The opacity of transparent objects can be held constant or can be varied according to the normal vectors to simulate light passing through more material near the silhouettes of the object. This is, of course, a more costly computation.

**novopacity** Transparency values constant over surface.  
**vopacity** Transparency values depend on surface normals.

The following is a possible transcription of a rendering session to illustrate the use of some of the shading options. (The “#” character indicates a comment.)

```
1 unix> getcmds render # Make sure the render aliases are loaded.
2 unix> show_defaults # This will show all the default switches.
3 unix> buffer vase.rle # Set a file name to redirect image to.
4 unix> render vase.a1 # Render the file vase.a1.
5 unix> onelight # Set some more options.
6 unix> white
7 unix> render vase.a1 # Render it again (previous image is replaced).
8 unix> flip # Turn vase inside out.
9 unix> render vase.a1 # One more time ...
10 unix> noflip # Turn it back right-side-out.
11 unix> state # What has changed from the defaults?
```

### Graphic Utility Switches for Render

Objects can be rendered on a black background or a colored background. The “background” command is followed by three integer numbers in the range 0-255 for the values of the red, green and blue channels.

**nobackground** Default background color (black).  
**background R G B** Set background color to given rgb value.

The **cull** command speeds up the rendering process by eliminating all back-facing polygons. If a scene is composed of only closed opaque surfaces, all back-facing polygons will be hidden and the **cull** option can be used without any change in the resulting image.

**nocull** Render all polygons.  
**cull** Throw out back-facing polygons.

When lines or curves are drawn with **render**, the program normally attempts to “miter” the corners between adjacent line segments. In some cases (where the line segments are very nearly parallel) this is undesirable, so the option is turned off by default. The effect will only be visible for relatively wide lines anyway.

**noneatcorners** Don't attempt to miter corners between line segments.  
**neatcorners** Try to make neat corners between line segments.

The **quilt** flag helps visualize the subdivision pattern that results from converting a B-spline surface into polygons. This option overrides the colors assigned the surfaces in the input files and assigns individual colors to each polygon by groups of four (i.e., there are four different colors). This results in a quilt-like pattern over the surface.

**noquilt**     Normal surface color.  
**quilt**        Indian blanket showing subdivision.

**Render** normally forms four triangles from each (nearly) flat surface region produced by the subdivision. It is possible to specify that only two polygons be formed, although this may result in shading that is not as smooth as usual.

**fourperflat**     Normal production of 4 triangles per flat.  
**twoperflat**     Form only 2 triangles per flat.

To continue the above example, here is a sample session to illustrate some of the general graphics options for the **render** program.

```
12 unix> defaults           # Reset all the defaults.
13 unix> background 100 100 100 # Set a grayish background.
14 unix> render vase.a1      # Render vase.a1 with the background.
15 unix> nobackground        # Turn off background.
16 unix> quilt              # Use quilt pattern.
17 unix> cull                # Throw out backfacing polygons.
18 unix> render vase.a1      # Render using quilt pattern.
```

### Output Switches for Render

The only form for the output of **render** is the run-length-encoded (RLE) file. This is a device independent format that can be used with various frame buffers. For example, the program **getfb** is a program that reads an RLE file and displays an image on a Grinnell frame buffer. Similar programs for other frame buffers are available, as well as an assortment of other programs for manipulating RLE files, as part of the Utah Raster Toolkit. The Utah Raster Toolkit is distributed with Alpha\_1 and documentation is included in a separate section of the (hardcopy) manuals. The image can be checked while it is being computed without any harm to the on-going **render** process. The **buffer** option should be followed by a file name. When **nobuffer** is used (the default), the output goes to stdout, and may then be redirected to a file or piped to another program. While redirecting the output of **render** to a file is strongly recommended, the use of pipes for **render** output is not generally recommended. It is better to create the RLE file and use that as the basis for further display or computation.

**nobuffer**     Output goes to stdout.  
**buffer filename**     Output is redirected to file (use full pathname).  
**buffer**        Output is piped to **getfb**.

**Render** outputs red, green, blue, and alpha channels to the RLE file. The alpha channel represents the percentage of coverage for each pixel. This channel is necessary for compositing images together, but may not be desired for a test render, because the RLE will be slightly larger with alpha. When using **filter** or **transp**, it is a good idea to have alpha output, because the image can be composited over other images nicely. Note that the **background** option sets **noalphaout**.

**alphaout**     Alpha channel will be output (default).

**noalphaout**

Alpha channel is not output.

The next two switches work together to produce a super-sampled filtered (anti-aliased) image. The **onlk** option computes the image at double resolution and the **filter** option then "averages" the image back to the original resolution. The **buffer** option may or may not be used, but is suggested since a 1k filtered image will take a considerable amount of time to compute. If **filter** is on but the **onlk** flag is not, then the resolution is rounded to a multiple of four and the image is filtered appropriately.

**nofilter** Normal point-sampled mode.  
**filter** Super-sampled and filtered mode. If you don't also specify **onlk**, you get a quarter-screen image.  
**offlk** Sample at normal resolution.  
**onlk** Sample at double resolution, **filter** is assumed.

Another way to do filtered images of better quality than the simple super-sampling scheme is with the a-buffer algorithm also available in **render**.

**noarender** Normal point sampled rendering mode.  
**arender** Render image with a-buffer filtering at each pixel.

The next pair of options allow the user to specify the image resolution. The defaults are 512X480 for the Grinell frame buffer.

**defxres** Use default x-resolution.  
**setxres X** Set x-resolution to X.  
**defyres** Use default y-resolution.  
**setyres Y** Set y-resolution to Y.

**Render** also has the ability to use different aspect ratios (i.e., the shape of a pixel) for various frame buffer monitors.

**defaspect** Resets aspect ratio to default (1.2).  
**aspect float** Sets aspect ratio to specified value.  
**matrix\_aspect** Sets aspect ratio to 0.954545 for matrix camera.  
**square\_aspect** Sets aspect ratio to 1.0 .  
**brl\_aspect** Sets aspect ratio to 1.25 .

Finally, the **viewport** option allows the user to render an image to an arbitrary rectangular viewport. The **viewport** command requires four integers in frame buffer coordinates for **xmin**, **xmax**, **ymin**, and **ymax**.

**noviewport** Set viewport to default (entire screen).  
**viewport x1 x2 y1 y2** Set viewport to specified (frame buffer, or screen space) coordinates.

Here, in our continuing example, are some uses of the output options.

```
20 unix> defaults          # Reset defaults again.
```



```

21 unix> buffer vase.rle      # Use RLE format.
22 unix> render vase.a1      # Render image to file.
                             # This can be done in the background
                             # so the user can do other things
                             # while the image is computing.
23 unix> getfb vase.rle      # Display image on frame buffer.
24 unix> filter
25 unix> onlk
                             # Compute double resolution filtered
                             # image (still using file vase.rle).
26 unix> render vase.a1      # Display filtered image.
27 unix> getfb vase.rle

```

### 16.1.2 Scripts For Rendering

The many options of the **render** program can be difficult to keep track of. To achieve consistent results, the use of Unix shell scripts is suggested. The shell script is a set of Unix commands grouped together in a file which can be executed by a single command. The user can set up a script to control the rendering process by putting the desired **render** option commands in the script file along with the commands to run the **render** program with the desired data. This way the user has more control over the process and can have a record of what happens. Also, the use of scripts allows the job to be batch processed during off-peak usage.

#### Getting the Correct Setup for the Script.

When a script is executed, a new C-shell is invoked and the commands in the script file are executed sequentially. In order to get the correct C-shell, the first character in the script must be the comment character "#". In addition, we usually begin scripts with "#!" which uses the first line to start up the C-shell. An example script might be:

```

#!/bin/csh -f
# This simple script will run /bin/csh with the -f option, which
# ignores the user's .cshrc file.
pwd
ls

```

It is important to note that in scripts the C-shell started is not an interactive one and will not normally source the Alpha\_1 shell variables and command files needed to run the **render** process. Therefore, it is necessary to do so explicitly in the script. The best way to do this is to include the line

```
source ~/alpha1/cmds/.a1defs
```

which sets up the basic set of commands which allow access to the other commands.

Any personal aliases or shell variables must be sourced by the script if they are to be used in the script.

#### An Example Shell Script

Here is an example script that might be used to render some data:

```

#!/bin/csh -f
# Use csh -f for fast startup.

# Since the C-shell started by script bypasses Alpha_1 setup,
# source the necessary Alpha_1 files.

```

```

source `alpha1/cmds/.aidefs

# Now the script's C-shell knows about getcmds, so
# load the render commands.
getcmds render

# Set the desired options.
onelight
white
background 100 100 200
buffer /tmp/vase.rle

# Run render. Be sure to use full pathname for the data files
# if they are not on the same directory that the script is run from.
# In this case the data file is on the user's data directory.
render /u/$user/data/vase.txt

# End of the script

```

### Executing a Shell-script

To summarize, there are three steps to using scripts to control the **render** process:

1. Create a script with your favorite text editor.
2. Make sure the script file is executable with the command:
 

```
chmod a+x script-file-name
```
3. Execute your script from the C-shell:
 

```
script-file-name
```

As executed above, the script will run in the foreground with output directed to standard output. The script can be run in the background with the output directed to a file with the following command:

```
script-file-name >& output-file&
```

This will redirect the output of the script's commands along with any error output into the file `output-file`.

## 16.2 Shading Parameters

This section discusses the input parameters to the **render** program that affect the shading process. These include light sources, transparency values, reflectance properties, and colors. Examples are given of parameter combinations that model certain materials such as metals (silver, gold, etc.) and plastic. The syntax used to specify these parameters is also described.

In most cases, appropriate **render** switches must be set in order for the shading parameters to be used. These switches are mentioned here but are described in detail in the section on render options (see section 16.1.1 [Render Options], page 216).

Finally, we need to make a distinction between global scene attributes and local surface attributes. Most of the shading parameters are global in that they apply to the whole scene and are not attached to specific surfaces (this may change in the future since it is desirable to have reflectance properties attached to surfaces). The only parameters that are currently surface attributes are:

<i>color</i>	The color (in red, green, blue coordinates from 0 to 255) of an object. The default if none is specified is a dull grey.
<i>back_color</i>	The color of the "back" side of an object (recall that all surfaces and polygons are oriented). The default if none is specified is a hot pink. (It is intended as a diagnostic tool for getting orientation correct.)
<i>opacity</i>	The value used for this object in transparent rendering. The default is 1.0 which signifies a completely opaque object.
<i>resolution</i>	The value used for deciding whether enough surface subdivision has been done to approximate the surface by polygons. The default value is 1.0, which is (very) roughly a measurement of straightness in screen space. This value produces images that look very good, and it should probably never be set lower than 1.0 (absolutely never below 0.5). For rougher images, values between 5.0 and 8.0 are normal, and for extremely coarse images, values up to 20.0 are reasonable.
<i>width</i>	If rendering polylines, the width parameter says how wide to make them. The default is 1.0, meaning one pixel wide lines.

For simplicity, all the global parameters can be gathered into one input file for use in rendering assorted data files (i.e., files with surface data in them). For example the parameters necessary for a metallic rendering of a surface might be put in a file called "metal.params" and the ones for a plastic rendering in "plastic.params". Therefore the user can set the necessary switches to render and render the surfaces either way without having to edit a data file.

### Lighting

Light sources are considered to be at infinity so that the vector from the object to the light source is a constant. This greatly decreases the amount of computation involved when compared to finite position light sources. The light source is specified simply as a vector pointing toward the light source. There may be up to ten such light sources specified in the input.

The light sources are interpreted in one of two ways by the render program. The **twolight** option to the **render** program treats the light sources as double light sources with the second source in the opposite direction of the specified light source. The **onelight** option treats the light sources as single sources.

Light sources are specified as follows:

```
light_vector = { x, y, z }
```

The x, y, and z are the coordinates of the vector pointing toward the light source.

If no light vector is found in the input a default vector of (1,1,1) is used which is up and to the right and behind the viewer.

### Transparent Rendering

Surfaces can be transparently rendered by adding an **opacity** parameter to each desired surface in the input. Opacity being the opposite of transparency, an opacity of 1.0 represents a solid opaque surface and an opacity of 0.0 represents a totally transparent surface.

The **opacity** parameter is a floating point number between 0.0 and 1.0. The default is 1.0 for totally opaque surfaces. The **opacity** parameter must be included as an attribute of a surface which is to be rendered transparently. This can be done most easily in **shape\_edit** when creating the geometry using the **setAttr** command. For example,

```
setAttr( Srf, "opacity", 0.1 );
```

To choose the **transparent** option the **transp** flag must also be set and normal interpolation used when running the **render** program.

A great deal of care must be exercised in choosing opacity values and colors which will blend well together. The transparency calculations are often counter-intuitive. There are two interactive programs available at Utah which help choose transparency values and colors. The **colors** program aids in choosing opaque colors and **tcolors** helps in the selection of colors along with opacity values.

### Reflectance Parameters

Last but not least are the parameters that describe how the surfaces reflect light. Remember that these are global parameters affecting all the surfaces rendered. First we will describe the different reflectance models available and then the parameters that control them.

The Alpha\_1 system makes use of three reflectance models: *cosine*, *Blinn*, and *Cornell*. The first is a model of diffuse and ambient reflections only, where the latter two are models of ambient, diffuse, and specular reflections. The differences between Blinn and Cornell models are subtle. The parameters which control them affect both in the same way, but identical sets of parameters may produce subtly different images. Cornell is the preferred model, and values suggested in this document are based on the Cornell model. *Diffuse* reflections are those in which the light from a given source is scattered equally in all directions when reflected by a surface. *Specular* reflections, on the other hand, are only reflections in the mirror direction (i.e., the highlights). Lastly, *ambient* reflections are those of ambient light which is equally present in all directions, not just from the given sources. The type of reflectance model used is chosen by setting the correct flag by issuing one of the following commands: **cornell**, **blinn**, or **cosine**.

### Blinn Parameters

The first control of the reflectance model is the percentage of contribution that each type of reflection makes to the total reflection. For example, the specular reflection is a large percentage of the total reflection for a metallic surface. These values, along with the refractive index of the surface, are specified by the parameters called **blinn\_params** using the **blinnParam** command in **shape\_edit**.

**blinnParam**( *Specular, Diffuse, RefractIndex* )

**Returns** <blinnParam> Creates an object representing the Blinn (and Cornell) shading parameters.

**Specular** <number:0-1> Percentage of specular reflection. Default is 0.3.

**Diffuse** <number:0-1> Percentage of diffuse reflection. Default is 0.4.

**RefractIndex**

<number> The refractive index is a positive number. For metals it is quite large (100-200), and for ceramics it is small (1-10). The default value is 100.

The percentage of ambient reflection is not specified since it is calculated from the other two so that they all sum to 1.0 (i.e.,  $\text{ambient} = 1.0 - (\text{specular} + \text{diffuse})$ ). The default is thus 0.3.

The name **blinn\_params** is a bit of a misnomer since these parameters can apply to either the Blinn or Cornell model. The cosine shading mode does not use these parameters. Cosine shading has a fixed percentage of 0.7 diffuse and 0.3 ambient.

### Distribution Parameters

The second control of the reflectance is the "shininess" of the surfaces or the shape of the highlight. This only applies to the Blinn and Cornell models which compute a specular component. Surfaces can be very "shiny" (have a narrow highlight) or can be "dull" (have a broad highlight).

This component of the reflectance model is described as a distribution function. This function is a

bell type curve that describes the distribution of microfacets on the surface of the object (i.e., the shininess of the surface). The distribution function can be defined as a single function or a weighted sum of separate distribution curves to model multiple surface roughnesses. The parameters are a weight and a slope and an object representing those parameters is created in `shape_edit` with the `distParam` command.

`distParam( Weight, Slope )`

**Returns** `<distParam>` Creates an object representing the distribution parameters for the Blinn and Cornell lighting models.

**Weight** `<number:0-1>` The weight assigned to this distribution in the weighted sum function.

**Slope** `<number:0-1>` The r.m.s. slope of the microfacets. A small number represents gentle slope (i.e., a smooth shiny surface), and a large number represents a rough surface.

Multiple instances of `dist_params` in a data file create a weighted sum distribution function (weights must add to 1.0). The default is a single distribution with a slope of 0.35.

### Modelling Materials

Metals are modeled as pure specular reflectors with a high refractive index. The `RefractIndex` should be set to 100 or greater; `Specular` should be set in the range of .9 to .95 and `Diffuse` should be set to 0. The choice of the percent `Specular` parameter depends on the color of the object and the desired contrast of the image. The higher the specular component, the higher the contrast. The `blinnParam` might be:

```
MetalBlinn := blinnParam( 0.95, 0.0, 200 );
```

For metals, the surface should be modeled with two sets of distribution parameters:

```
MetalDist1 := distParam( 0.4, 0.4 );
MetalDist2 := distParam( 0.6, 0.2 );
```

Since metals are homogeneous materials, the `nowhite` flag should be set and multiple light sources are recommended.

The color of the surface for a metallic rendering is very important. Technically, the color is dependent on the monitor and color map used. The following table gives some pre-computed colors (rgb) for various metals. These were computed for the Barco monitor but will give reasonable results on other systems. (The file "metals.a1" on the Alpha\_1 \$data directory contains these colors.)

polished bronze	174 145 81
polished copper	199 135 92
polished silver	234 252 222
aluminum	229 253 227
gold	191 142 57
platinum	194 201 165

Plastics are modeled as nonhomogeneous materials that are highly diffuse reflectors. The refractive index is set high; the specular is set in the .1 to .2 range; the diffuse in the .6 to .8 range. The `blinnParam` might be:

```
PlasticBlinn := blinnParam( 0.2, 0.7, 100 );
```

One set of distribution parameters is used:

```
PlasticDist := distParam( 1.0, 0.15 );
```

Since plastics are not homogeneous, the **white** flag is set. Single lights are usually sufficient for plastic surfaces but multiple lights may be used. When using multiple lights with highly diffuse objects, one is cautioned to lower the diffuse component to avoid mach bands. Plastics can, of course, be any color; however darker, more saturated colors give better results.

### 16.3 Render Flags

This section contains a quick reference to the commands that set switches for the **render** program (see section 16.1.1 [Render Options], page 216 for detailed descriptions). Individual options may be chosen by executing any of the commands in the left column below. Brief descriptions of their actions are in the right column. Groups of commands indicate switches (only one option in any group is on at any particular time). The default appears first in each group.

#### Shading Options

<b>cornell</b>	Cornell lighting model.
<b>blinn</b>	Blinn lighting model.
<b>cosine</b>	Cosine lighting model.
<b>normal</b>	Use normal interpolation.
<b>flat</b>	No shade interpolation across polygons.
<b>smooth</b>	Shade interpolation across polygons.
<b>dynamic</b>	Set up for using dynamic shading program.
<b>nopseudocolor</b>	No color interpolation (use normal shading).
<b>pseudocolor</b>	Color interpolation across polygons with no shading.
<b>noflip</b>	Leave mesh orientations as they are.
<b>flip</b>	Reverse the orientation of all surface meshes.
<b>fasthighlights</b>	Viewer at infinity.
<b>goodhighlights</b>	Viewer at finite distance.
<b>nowhite</b>	Highlights same color as object .
<b>white</b>	White highlights (blinn/cornell only).
<b>twolight</b>	Double light source.
<b>onelight</b>	Single light source.
<b>notransp</b>	All objects rendered opaque.
<b>transp</b>	Turn on transparent rendering and normal interpolation, only useful if the objects have transparency values.
<b>novopacity</b>	Transparency values constant over surface.
<b>vopacity</b>	Transparency values depend on surface normals.

#### Graphics Utilities

<b>nobackground</b>	Default background color (black).
<b>background R G B</b>	Set background color to given rgb value.

**nocull**      Render all polygons.  
**cull**        Throw out back-facing polygons.  
**noquilt**     Normal surface color.  
**quilt**       Indian blanket showing subdivision.  
**fourperflat**      Normal production of 4 triangles per flat.  
**twoperflat**    Form only 2 triangles per flat.  
**norendercombine**    Do not try to interpret boolean combination expressions in the file.  
**rendercombine**    Interpret combiner objects in the file by performing the boolean combinations on a pixel-by-pixel basis. Currently this is only implemented for expressions that form closed objects. This is fairly expensive, so **xres** and **yres** should be used to limit the computation to a smaller screen area. It is intended as an aid in checking data for the **combine** program, not as a replacement for it.

### Output Options

**nobuffer**    Output goes to stdout.  
**buffer *filenm***    Output is redirected to file (use full pathname).  
**buffer**      Output is piped to **getfb**.  
**alphaout**    Alpha channel will be output.  
**noalphaout**    Alpha channel is not output.  
**nofilter**    Normal point-sampled mode.  
**filter**      Super-sampled and filtered mode. If you don't also specify **on1k**, you get a quarter-screen image.  
**off1k**       Sample at normal resolution.  
**on1k**        Sample at double resolution, **filter** is assumed.  
**noarender**    Use normal point-sampled filtering.  
**arender**     Use a-buffer filtering at each pixel.  
**defxres**     Use default x-resolution.  
**setxres *X***    Set x-resolution to *X*.  
**defyres**     Use default y-resolution.  
**setyres *Y***    Set y-resolution to *Y*.  
**defaspect**    Resets aspect ratio to default.  
**aspect *val***   Sets aspect ratio to specified value.  
**matrix\_aspect**    Sets aspect ratio to 0.954545 for matrix camera.  
**square\_aspect**    Sets aspect ratio to 1.0 .  
**brl\_aspect**    Sets aspect ratio to 1.25 for BRL.  
**noviewport**    Set viewport to default (entire screen).  
**viewport *x1 x2 y1 y2***

Set viewport to specified (fb) coordinates.

## 16.4 Render Parameters

This section is a reference card describing the syntax and default values for input parameters that affect the shading in the **render** program (see section 16.2 [Shading Parameters], page 222 for detailed descriptions).

The syntax is presented along with brief descriptions of what the values are. The default values are given and any switches that affect the parameter are listed.

### Lighting

```
light_vector = { x, y, z }
```

The value is the coordinates (integer or floating) of the vector pointing toward the light source. The default value is {1, 1, 1}. Relevant **render** switches are **onelight** and **twolight**.

### Transparent Rendering

Set transparency values as in:

```
setAttr( Srf, "opacity", 0.1 );
```

The value is a floating point number between 0.0 and 1.0. The default is 1.0, producing totally opaque surfaces. Relevant **render** switches are **transp**, **notransp**, and **normal**.

### Reflectance Parameters

Create Blinn shading parameter objects with

```
DefaultBlinn := blinnParam( 0.3, 0.4, 100 );
```

The refractive index (last argument) is a positive number which is large for metals (100–200) and small for ceramics (1–10). The percentage of specular and diffuse reflection (first two arguments) are floating point numbers between 0.0 and 1.0. Default values are as in the above example. Relevant rendering switches are **cosine**, **blinn**, **cornell**, **normal**, **flat**, **white**, and **nowhite**.

Create distribution parameter objects with

```
DefaultDist := distParam( 1.0, 0.35 );
```

The weight (first argument) is the weight assigned to this distribution, a floating point number between 0.0 and 1.0. The slope (second argument) is also a floating point number between 0.0 and 1.0 which gives the r.m.s. slope of the microfacets. A small number represents a gentle slope (i.e., a smooth shiny surface), and a large number represents a rough surface. The default is one set of **distParams** with values as in the example. Relevant **render** switches are **normal**, **white**, **nowhite**, **blinn**, and **cornell**.

## 16.5 RLE Utilities

As mentioned before, the **render** program produces a run-length encoded form of a raster image. There are many tools available for operating on these raster images stored in RLE files. These programs can be piped together at the Unix shell level to produce a wide variety of images.

A separate set of documents is provided with this manual that describes the Utah Raster Toolkit, a large collection of tools for dealing with RLE image files. This toolkit is distributed with Alpha\_1.



## 17. Other Display Options

This section describes other display utilities which are available in addition to the high quality raster rendering. These include generating hidden-line images and using fast polygon rendering devices if they are available.

### 17.1 Making Polygons

Some of the alternate display programs described in this chapter require polygon data as input, rather than surface data. Polygonal data is also frequently used as an interface to other programs which are not part of Alpha\_1 (and don't understand spline surfaces). There are three Alpha\_1 programs which produce polygons: **dsurf**, **srf\_mash**, and **tess**. The **dsurf** program is actually just the **render** program compiled separately with a special flag so that polygons are dumped to a file when they are created rather than being drawn on the screen. In general, **dsurf** is not the best way to make polygons, but it contains some options which are not available in the other two programs which may be required (especially for producing hidden line drawings). The **srf\_mash** program is similar to **dsurf** in the algorithm which is used to subdivide surfaces, but it does not operate in scanline order or clip away portions of the surfaces which "fall off" the screen. The **tess** program is a fast tessellator, which uses a single-pass refinement estimation scheme for much greater speed than **dsurf** or **srf\_mash** for similar results. Both **dsurf** and **srf\_mash**, like **render**, use an adaptive refinement which divides a surface into two parts, checks whether each half is close enough to flat to be represented as a polygon, and repeats the process if not. The scheme used in the **tess** program analyzes the surface, using curvature estimates to decide how many polygons should be used to represent the result, and makes a single (intensive) refinement to generate the new control points needed.

The **dsurf** command is loaded when the standard **render** commands are loaded using the "render" keyword to **getcmds**, as in "getcmds render".

**dsurf files**

**Output**      <binfile> A binary file of polygons which approximate the input data.

**files**        <a1files> The files which are to be converted to polygons.

**Dsurf** accepts all globally-known objects and passes them straight through, with the exception of transformations (**view\_trans**) and surface descriptions (**dsurf**). The former are used to map objects into "lighting space" and then deleted, and the latter are subdivided until "flat" and output as polygons.

The following commands which affect the **render** program are also relevant for **dsurf**: **flip** and **noflip**, **quilt** and **noquilt**, **noviewport** and **viewport**, **onlk** and **offlk**, **twoperflat** and **fourperflat**, **neatcorners** and **noneatcorners**. For hidden line generation using the **hidden** program (described later in this chapter), one additional flag called **adj** generates adjacency information about neighboring polygons in the output.

**noadj**        Do not generate adjacency information between polygons in the output.

**adj**          Generate adjacency information between polygons.

The commands for **srf\_mash** are loaded with the "srf" keyword to **getcmds**. The **render** commands **noquilt**, **quilt**, **twoperflat**, and **fourperflat** are relevant to **srf\_mash** as well.

**srf\_mash files**

**Output** <binfile> A binary file of polygons which approximate the input data.  
**files** <a1files> The files which are to be converted to polygons.

Finally, the **tess** program commands are loaded with the "tess" keyword to **getcmds**. The following commands affect the output of the **tess** program.

**notessquilt** Polygons inherit the color of the original surface.  
**tessquilt** Assign alternating colors to the polygons so that the tessellation can be seen when the polygons are rendered.  
**deftessres** Default resolution for surfaces is 50.0, which should produce a fairly close approximation to the surface, roughly analogous to a resolution of 1.0 in the rendering program. Larger values give rougher approximations (fewer polygons).  
**tessres res** Set the surface resolution to the given value.  
**notessnoref** Use the surface resolution value to control refinement.  
**tessnoref** Do no refinement (essentially sets the resolution to infinity).  
**tessgoodnorms** Use a better technique for estimating normals (but slightly more expensive).  
**notessgoodnorms** Use a simple normal estimation technique.  
**notessquads** Always divide quadrilateral surface pieces into two triangles.  
**tessquads** Put out quadrilateral surface pieces as quadrilaterals. (Warning: do not try to use the output for **render** - the shading algorithm in **render** makes some assumptions that do not hold for this kind of output.)  
**notessmaybequad** Do not put out quadrilaterals conditionally.  
**tessmaybequad** Put out quadrilateral surface pieces as quadrilaterals if they are really flat (within floating point fuzz). (Warning: This is probably risky as **render** input, although not as bad as always making quadrilaterals would be.)  
**tessnormnorm** Normalize the normal vectors associated with polygon vertices.  
**notessnormnorm** Don't normalize the normal vectors.  
**notessholefill** Don't attempt to fill "cracks" or "holes" that may appear between surfaces.  
**tessholefill** Some internal subdivision done by the **tess** program may result in small "cracks" between the polygons approximating a single surface. These may be removed using this flag, but it is somewhat expensive.

The **tess** program itself accepts the usual list of files, and produces polygons in binary format.

**tess files**

**Output** <binfile> A binary file of polygons which approximate the input data.

*files*            <allfiles> The files which are to be converted to polygons.

## 17.2 Hidden Line Elimination

The **hidden** program produces a set of polylines which when displayed create a hidden line drawing of the set of Alpha\_1 objects in the input stream. The **hidden** program is basically a polygon hidden line algorithm. Surfaces to be drawn will generally need to be preprocessed to produce a set of polygons which approximate the surface. The **hidden** program processes these polygonalized surfaces by assigning edge types to the edges of the polygons. The edge types indicate which part of the surface a polygon edge corresponds to (boundary, silhouette, shared, etc.) The **adj** flag to **dsurf** tags the output polygons with adjacency information, so that **hidden** is able to type the edges correctly.

The various options which can be set for **hidden** affect the type of output produced. All surface data (except when **mesh** is turned on, see below) must first be run through a preprocessor such as **dsurf** or **tess** to turn surfaces into polygons. The **adj** flag is currently only available for **dsurf**, not for **tess**, so better hidden line drawings will come with **dsurf** as the preprocessing step.

The **hidden** commands are loaded with the "hidden" keyword to **getcmds**, as in "getcmds hidden".

**hidden files**

*Output*            <binaryfile> A set of polyline objects which represents the hidden line view of the input data.

*files*             <objectstream> The geometric objects from which hidden lines are to be removed.

An example sequence of commands for **hidden** might be as follows. (The **dsurf** commands are loaded with the **render** commands.)

```
getcmds hidden
getcmds render
adj
dsurf file1,file2,file3 >file.polys
hidden file.polys >file.hid
```

The following mode switches are available. The defaults are given first for each switch.

**nodbg**

**dbg**             Visual debugging showing edge typing.

**nobounds**

**bounds**        Normally, lines of surface polygonalization are produced. With **bounds**, only surface silhouette lines and surface boundaries are part of the output.

**nomesh**

**mesh**           If on, produces the set of visible lines of a surface control mesh. The input may contain surfaces with this option. The results are not always acceptable because the quadrilaterals of the control mesh can be very non-planar.

**nozkluge**

**zkluge**        If on, moves boundary & silhouette lines forward a bit in Z. This makes data which is going to be rendered look a little better.

**depsilon**

**epsilon e** The default epsilon for approximate equality checks is 5.0e-3, but can be reset with the **epsilon** command.

To set all the default values for the program use:

#### **hiddefaults**

**Output** <Nil> Set all the switches for the **hidden** program back to their default values.

#### **show\_hiddefaults**

**Output** <textfile> List all the default values for **hidden** program switches.

#### **hidstate**

**Output** <textfile> List all the switches of the **hidden** program which are currently set to values other than the default.

The **quick** program (see section 17.4 [Fast Line Rendering], page 234) is a fast way to view the output of the **hidden** program. The **view** program also works, and is available on many more displays.

#### **Style files**

It is possible for **hidden** to assign different color and line width attributes to the different types of lines output. This has been implemented with a list of string attributes which can specify (1) whether or not to draw a particular type of line, (2) what color to assign to a particular type of line, and/or (3) the line width of a particular type of line. The default line color is white and the default line width is one, so a style file needs to be used only if you would like to change these defaults or have hidden lines in the data set displayed.

The eight types of lines for which attributes can be set are:

```
boundary
hid_boundary
silhouette
hid_silhouette
shared_edge
hid_shared_edge
interior_line
hid_interior_line
```

"Boundary" lines are those which come from surface boundaries. "Silhouette" lines are lines in the particular view where the surface bends away from the viewer. For example, a cylinder has two lines up the sides (connecting the top and bottom disk) which are silhouette lines rather than boundaries of the surface. "Shared edges" are internal polygon edges which do not occur on surface boundaries (so named because they are always shared by more than one polygon). "Interior" lines are lines which might have been mapped onto the surface. They are not currently supported by the **hidden** program.

To create attributes from **shape\_edit** which can be put into the input stream for **hidden** to toggle output of any of the line types, use something like:

```
Attr := attribute( "hid_boundary", "TRUE" );
Attr := attribute( "hid_boundary", "FALSE" );
```

where "TRUE" indicates that that line type (in this case **hid\_boundary**) should be output.

To assign a color to a line type, create an attribute with a string with the color values. The name should have “\_color” appended to the line type. Similarly, to assign a particular width to a line type, append “\_width” to the attribute name.

```
ColAttr := attribute( "hid_boundary_color", "255 255 0" );
WidAttr := attribute( "hid_boundary_width", "0.1" );
```

These attributes can have a global effect if they are at the beginning of the input stream to hidden. If they follow a “srf\_name” attribute in the input stream, they will affect only those polygons which came from a surface containing a corresponding “name” attribute.

```
RestrictAttr := attribute( "srf_name", "this_one" );
...
setAttr( Srf1, "name", "this_one" );
```

### 17.3 Fast Polygon Programs

There are currently two programs for rendering surfaces and polygons on the Lexidata Solidview display. As more polygon rendering engines are supported, these programs should be generalized to run on other devices, just as view runs on several devices.

The aliases for both programs are loaded using the “lexi” keyword to **getcmds**, as in “getcmds lexi”. This will also cause the **render** commands to be loaded if they have not already been, as some of the switches for the standard **render** program are also meaningful for the Lexidata rendering programs. In particular, the shading parameters affect the rendering on the Lexidata. Other switches may or may not be used by the Lexidata rendering programs.

The **lexiview** program accepts only polygon input — surfaces must be run through the **dsurf** program before being fed to **lexiview**. The program transforms all the polygons in the input stream to screen space, applying any transformations in the input stream, and displays them on the lexidata according to specified shading modes. Transparency flags and values may be specified.

**lexiview files**

**Output**      <Nil> Display the input on the Lexidata display.  
**files**        <objectstream> The data to be displayed, should be polygons.

An example sequence of commands for Lexidata display might be:

```
dsurf foo.mat,foo.a1 >foo.polys
lexiview foo.polys
```

or

```
dsurf foo.mat,foo.a1 | lexicmd
```

The **lxrender** program accepts either surface or polygon data, although surface data is more meaningful. **Lxrender** produces a rough image of a surface almost immediately, and gradually refines the surfaces providing an image of steadily improving quality. Surfaces are turned into polygons as soon as they are read in, and these polygons displayed as the first rough image. The surfaces are then subdivided once, and the next set of polygons displayed. This process continues until a specified number of subdivision levels have been reached. Note that the adaptive subdivision which is used for the standard rendering program is **not** in effect in **lxrender** (although it would be a nice extension if it were, and would significantly improve the effect of **lxrender**). The subdivision is a pure binary tree, with the number of levels in the tree determining the quality of the final image.

**lxrender files**

**Output** <Nil> Display the input on the Lexidata display.  
**files** <objectstream> The data to be displayed, usually surfaces.

The **lxrender** program allows the following options to be specified:

**defllevels**  
**lxlevels** *n* <integer> The default number of subdivision levels is 5. The value can be set as desired, but the processing time goes up exponentially. So a level of 8 is takes a lot longer than a level of 5.

**deflcolors**  
**lxcolors** *n* Declare that space is needed for *n* colors (default is 16). Note: The shading quality degrades as the number of colors increases. Sixteen colors allows the full 8 bits of shading information, while 64 colors would allow only 6 bits of shade information.

**nolxverbose**  
**lxverbose** Talkative mode, prints message as it starts new levels. This is useful for large datasets where you want to know that it is really thinking.

**nolxquick**  
**lxquick** Quick(er) mode, only displays final stage. In normal mode, the polygons at each subdivision stage are displayed.

**nolxmeshes**  
**lxmeshes** By default, polygons are formed from the points only at the corners of the control mesh. When specified, the meshes option will form polygons for each facet of the control mesh. For **nolxmeshes**, the **smooth** shading switch must be set. For **lxmeshes**, the **flat** shading switch must be set.

**Lxrender** also pays attention to the **background** command as for **render**, but no transparency capability is provided.

Bug note: Things seem to go wild if **smooth** is not turned on at the same time as **nolxmeshes**, or if **flat** is not turned on with **lxmeshes**.

## 17.4 Fast Line Rendering

The quick program provides a faster rendering display than the **render** program for line data. It displays lines, polylines, and polygon (boundaries) on either the frame buffer or the lexidata. Curves are displayed by refining them and displaying the resulting control polygon and surfaces are displayed by converting the control mesh to a set of quadrilaterals and displaying those polygons. Note: The **view** program will also accomplish this purpose, and is probably just as fast. The quick program was built at a time when the output of **hidden** was not compatible with **view** and the only way to display the results was with the (relatively slow) **render** program.

The commands are loaded via the "quick" keyword to **getcmds**, as in "getcmds quick".

### quick files

**Output** <Nil> Display the input on the Grinnell frame buffer or Lexidata.  
**files** <objectstream> The data to be displayed, should be "line" data.

The following mode switches are available:

**quickfb** Display on the frame buffer.

**quicklexi**    Display on the lexicdata.

**defquickaspect**  
**quickaspect f**  
               Set aspect ratio (when **quicklexi** is turned on, this is set to 1.0).

**quickclear**  
**noquickclear**  
               Normally **quick** clears the screen before drawing, but it can be turned off with this switch.

**quicktran**  
**noquicktran**  
               Normally, input data is transformed to screen space, but if **noquicktran** is set, no transformations applied.

**noquickclip**  
**quickclip**    Turn on (a rather cheap) clipping option. It slows things down, and does not do any kind of Z clipping, but does give the same shape as render will.

**noquickoverlay**  
**quickoverlay**  
               Draw the image into the overlay planes on the frame buffer.

## 17.5 Ray Tracing

Ray is a ray tracing program that renders high quality scenes from geometry generated by Alpha\_1. (Note: The actual name of this program is "prt", but all the commands reference it as "ray". The "prt" name will probably disappear eventually.) The **ray** program does all of the wonderful things ray tracing is famous for: shadows, reflection, refraction, texture mapping, motion blur, anti-aliasing, penumbras, diffuse reflections and more.

Ray is relatively quick as ray tracing programs go. However, **ray** is **not** an interactive program. Its place in the design loop is towards the end, when a high quality image is needed to best represent the work. **Ray** is best at rendering scenes, not objects. It doesn't make sense to do shadow testing if there's no floor to cast a shadow on, reflections make little sense if there's only one object in the scene. Getting good pictures out of **ray** usually requires additional modeling time as well as extensive CPU time.

### 17.5.1 Data Preparation

#### Subworlds

The **ray** program uses hierarchical bounding boxes to reduce the search space while tracing rays. If a box is placed around a number of objects, **ray** can test against this box before investigating each one individually. For scenes with a large number of objects, this can speed it up significantly. Groups of objects around which these boxes can be formed are called *subworlds*.

In order to group objects into a subworld, they are placed in a group with a **subWorld** attribute. In **shape\_edit**, the **subWorld** command attaches this attribute to a group of objects.

**subWorld( objects )**

**Returns**    <Nil> Sets the subworld attribute on a group.

**objects**    <group> The group of objects which will make up the subworld.

### Optical Characteristics

Using **ray** differs in several ways from using **render**. Unlike **render**, each individual object can have different surface qualities (specular, diffuse reflection, etc). These parameters are represented with an object in **shape\_edit** which can be created with the **surfaceQuality** function.

**surfaceQuality**( *Color, RIndex, Specular, Diffuse, Transmit, Reflect, GExp, GDiff* )

**Returns** <surfaceQuality> Create an object with the given parameters.  
**RIndex** <number> Index of refraction.  
**Color** <listOf number> Like the colors used for **render**.  
**Specular** <number> Specular reflection component for Phong shading model.  
**Diffuse** <number> Diffuse reflection component for Phong shading model.  
**Transmit** <number> Light passing through the object (refraction component).  
**Reflect** <number> Light reflecting off the object (reflection component).  
**GExp** <number> Glossy exponent. For Phong shading model, this specified the specular exponent.  
**GDiff** <number> Glossy diffusion. The amount reflections are to be spread out. Produces effects like brushed metal or frosted glass. For a unit length reflection or refraction ray, it sets the radius which the end of the ray may be perturbed by. The numbers are typically small - less than 0.1.

The name of the object will be used to reference it from surfaces which have these parameters via an "optics" attribute. For example,

```
Dull := surfaceQuality( 1.2, list( 255, 239, 230 ),
                        0.0, 0.96, 0.0, 0.0, 1.0, 0.0 );
setAttr( Object, 'optics', "dull" );
```

If an object has no optics attribute, it is given a default flat white (a la **render**), but the **ray** program complains about the lack of optics information for the object.

### Texture

**Ray** supports texture mapping surfaces with RLE files. A textured object has the RLE file mapped exactly across the parametric range of the surface. The colors at a particular point on the surface are used in place of the color in the surface's optical information. If the texture's RLE file has an alpha channel, it is used by **ray**. Rays will pass through areas of alpha = 0 as though it was invisible, and will hit areas of full coverage (alpha = 255) normally. Interestingly enough, this presents a way to render trimmed surfaces. By rendering the trimming curve as a polygon in parameter space and using the resulting image as a texture, the rest of the surface is cut away.

Texture mapping is simple to use. Textures used in the ".a1" file can be defined ahead of time with a global "texture" attribute. The objects to be mapped with this texture have a similar attribute. If a texture map only has one channel (black and white), then the resulting color is the surface's color (as specified in the **surface\_quality** object) modulated by the texture color. To get woodgrain for example, all you need to do is map a black and white wood texture onto a light brown object. (The global attribute is no longer required, so leave out the first line if you wish.)

```
TextureAttr := attribute( "texture", "texture_file_name.rle" );
setAttr( Object, "texture", "texture_file_name.rle" );
```

### Global Viewing Environment

**Ray** uses the view matrices generated by **view** for object orientation. It does not, however, recognize render-style screen transformation matrices (because there is no such thing as a "perspective transformation" in ray tracing). **Ray** does not concatenate multiple view matrices.



**Ray** uses the same perspective as the PS300 and Xgen **view** programs. The eye is five units out from the origin, with the screen ranging from -1.0 to 1.0. This can be changed with two global attributes which can be easily created in **shape\_edit**. The **view\_distance** attribute specifies the distance from the eye to the origin, and the **view\_size** attribute specifies the range of object space covered by the screen (symmetric about 0.0).

```
ViewDist := attribute( "view_distance", 5.0 );
ViewSize := attribute( "view_size", 1.0 );
```

**Ray** uses a fixed aspect ratio of 1.0; use the **fant** program to deal with frame buffers that have other aspect ratios.

Light sources in **ray** are quite different from **render** — they are defined in object space, not in screen space. In other words, they are specified as a three dimensional point in space, not a unit distance vector. The default light position is (10, 10, 100). It can be set by creating a **lightSource** in **shape\_edit**.

```
lightSource( Location, Intensity, Radius )
```

**Returns** <lightSource> An object representing a light source (currently only used in the ray tracing program).

**Location** <point> The location of the light source in space.

**Intensity** <number> The intensity, a number between 0.0 and 1.0.

**Radius** <number> Radius of the light source. This is used for creating penumbras. If the penumbras are too wide, then PRT will find the shading to be different for each pixel. This causes the thrasher to super-sample each pixel, and the image will run extremely slowly.

Ambient light can be supplied with a "light\_ambient" attribute:

```
AmbientLight := attribute( "light_ambient", 0.2 );
```

Lights that don't cast shadows are useful because shadow testing is one of the most expensive things the ray tracer does. This allows you to add lights for bringing out highlights without paying for a shadow on each one. To get this, put a "no\_shadow" string attribute onto the light.

## 17.5.2 Invoking Ray

The **ray** program reads a binary Alpha\_1 data stream on standard input. It generates its output to a file specified. It is usually a good idea to do the command:

```
unlimit
```

on Unix 4.2 BSD systems before running **ray**. Because **ray** must maintain the entire set of subdivided polygons in core, it often has a large working set.

**ray files**

**Output** <Nil> Creates an image in the file specified with **raybuff**.

**files** <objectstream> The data to be rendered.

The various switches for the **ray** program are described below. The first one in each group is the default.

**defraybuff**

**raybuff file.exp**

Specifies the name of the file for the ray traced output. If you forget to

specify a file, the output will end up in file "rayout.exp". The file is an RLE file, but it is in "exponential" format. This format stores an extra fifth channel (in addition to R, G, B and coverage) containing an "exponent" for the red, green and blue values. This allows a larger dynamic range. Conversion from the ".exp" file to a normal RLE file is done with **unexp** (see below). Output from **ray** always has a proper coverage channel.

**norayexpmax**

**rayexpmax** *exp*

Set the exponent (maximum pixel value) for the output. The RLE file produced is a normal one rather than the default in "exponential" format.

**noraylog**

**raylog**

When logging is on, **ray** generates messages to stderr indicating its progress. (See below for more details.)

**norayfilter**

**rayfilter**

Enable filtering. Without this flag, no filtering or pixel threshing is done. You must specify **rayfilter** in order to use **rayjitter**, **raythresh**, **raygauss**, **raygrid**, or **raysample**.

**norayjitter**

**rayjitter**

Enables spatial jittering. Useful for penumbras and diffuse reflections to avoid contouring. The results from jittering are usually rather distracting unless a large number of samples are taken. Note: Jittering in the time domain is controlled by the **frames** program, not by **ray** (see the section on motion blur below).

**noraygauss**

**raygauss**

Use a gaussian filter instead of a box filter. This is most useful when a very high quality image is made with a large number of samples. The "bell curve" of the gaussian is 2.0 pixels wide, ranging from the centers of the adjacent pixels. It helps reduce "roping" effects on nearly horizontal or vertical edges. For best results, the sampling width (**raysample**) should be about 1.5 to 2.0.

**defraythresh**

**raythresh** *lines*

Specifies the number of lines used by the pixel thresher. **Ray** keeps a buffer *lines* scanlines of previously rendered pixels. When adjacent pixels are different, **ray** uses super-sampling on these pixels to try and find more detail. If this causes the pixel values to change, it compares other pixels — including those in previous scanlines — to look for more possible changes. The **raythresh** switch allows you to specify how many scanlines it searches back. The default is three lines; more may be needed if small, isolated details are present in the image.

**defraygrid**

**raygrid** *size*

Specifies the per-pixel sampling grid size. A grid size of *N* performs *N*-squared samples per pixel. The default is five (25 samples per pixel).

**defraysample**

**raysample** *width*

The width of the sampling area in pixel units. A sampling width of 1.0 (the default) places the samples evenly over the pixel. A width of 2.0 places

the samples over an area ranging from the centers of the adjoining pixels. Values much larger than 2.0 will cause the image to blur (particularly if -g isn't used).

**defraysize**

**raysize rastersize**

Gives the raster size for the image. Default is 512.

**defrayheight**

**rayheight start\_y end\_y**

Gives the starting and ending scanlines for the image. Default is to go from 0 to the rastersize.

**defraywidth**

**raywidth start\_y end\_y**

Like **rayheight**, for the width of the image.

For images that take a non-trivial amount of time, this **raylog** switch is useful to monitor progress. After surface subdivision and preprocessing, the standard error from **ray** contains lines of the form:

```
/* Scanline L: ST, total TT, super-sampled N + M */
```

These numbers are:

L	Current scanline number.
ST	CPU time used for that scanline.
TT	Cumulative CPU time used.
N	Number of pixels super-sampled on this scanline.
M	Number of pixels super-sampled on previous scanlines for this scanline (your pixel thrasher at work).

### Choosing the Image Quality

The number of options **ray** has for filtering and super-sampling allows you to gradually trade CPU time for image quality. Although there is much room for research to find the optimum parameters, the following suggestions can be used as a general guide.

With no options set, as in:

```
ray file1,file2
```

no anti-aliasing is done, so the picture is very coarse, but will finish in a pretty reasonable amount of time. This is only useful for debugging, e.g., to see if textures are oriented correctly. Also see the section about **Xray** below.

A slightly better image results from

```
rayfilter
ray file1,file2
```

This will use three thrasher lines, and a sampling grid of 5x5 samples per pixel over exactly one pixel area. The image is "OK" but not perfect if you get close, and it will finish in a finite amount of time.

A really good image might use

```
rayfilter
raygauss
raygrid 8
```

```

raysample 2.0
raythresh 6
ray file1,file2

```

This takes 64 samples per pixel (for pixels that are super-sampled) over an area of two pixels. The results are filtered with a gaussian filter. The images look great (just slightly better than the **arender** option of **render**), but will not finish in a finite amount of time unless some arrangement for providing a large number of CPU cycles is made.

### The Xray Program

There is another version of **ray** that is interfaced to the X window system. It is quite useful for ensuring that **ray** is on the right track, since some things (like texture maps) can't be previewed with **render**, but output goes to an X window rather than a file.

#### Xray files

```

Output    <Nil> Creates an image in an X window.
files     <objectstream> The data to be rendered.

```

The same switches can be use as for **ray**, plus the additional **xraymag** switch.

#### defxraymag

##### xraymag keyword

Specifies magnification at which the image will be viewed. Possible values are the words "one", "two", and "three".

An example might be:

```

xraymag two
raysize 256
xray file1,file2

```

A window 512x512 pixels wide is generated, with each **ray** pixel shading four screen pixels. This allows a closer look at problem areas. The **Xray** program always converts the image to black and white, so the results will look reasonable on a workstation with 8 bits/pixel or more. Before shading the image, **xray** renders the subdivided polygons with lines — so you know what to expect before it starts tracing rays.

### The Unexp Program

The output **ray** generates is normally an RLE image stored in "exponential" format. These files contain a fifth channel to store the exponent for the red, green and blue channels. This expands the dynamic range of the possible pixel values. The **unexp** program converts this exponential file to a straight RLE file. This program is part of the Utah Raster Toolkit, so refer to the documentation (at the back of this manual) for a full description.

### On Using Motion Blur

**Ray** supports motion blur in conjunction with the **frames** program. Since having **ray** perform the transformations on moving objects would be prohibitively expensive, the rays are instead moved to the objects. In order to accomplish this, each moving subworld has a number of blur matrices attached to it. **Frames** generates one matrix for each position in the pixel subsample grid, and (optionally) additional matrices within each of these times to randomly choose from. From a user's point of view, this means that the size of the subsampling grid (**raygrid** option) must be the same for both **ray** and **frames**. **Ray** automatically detects cases where it is reading motion blurred data.

## 17.6 Hardcopy With PostScript

A utility program for converting Alpha\_1 text data (".a1" files) into PostScript files for printing on the Apple LaserWriter (or any device that understands PostScript) is called **alps**. The program pushes lines, polylines, polygons, curves and surfaces through any viewing transformations in the input stream and through the perspective transformation. Lines and polylines go into the output stream as is. Curves and surfaces are turned into polylines which represent control polygons and meshes, smooth curves, or isoparametric lines using the same routines that **view** now uses. The output form is controlled by the flags described below. The **view** program should be used to position the objects to be printed; **alps** expects the data to be in the usual [-1,1] space. The program attempts to make some intelligent adjustments for the positioning of the [-1,1] box on the page, centering it horizontally and vertically as appropriate. **Alps** does set a clipping box to the [-1,1] boundary, so you will not see parts of the image that lie outside this area.

The commands for using **alps** are available via the "alps" keyword to **getcmds**, as in "getcmds alps".

**alps files**

**Output**      <postscriptfile> A PostScript file which will generate an image of the input data.

**files**        <objectstream> The objects that should be in the image.

Typical usage might be:

```
unix> alps obj.mat,obj.a1 >obj.ps
unix> lpr -Plw obj.ps
```

Switches available for **alps** are listed below. The first one in each pair is the default.

**defpsheight**

Use the default **alps** height.

**psheight float\_value**

Set the height of the figure.

This sets the height in inches which the figure is to occupy on the page. The default is normally 11 inches, unless scribe mode is specified (see below) in which case the default is 9 inches because the usual top and bottom margins in scribe are 1 inch.

**noscribemode**

Don't assume the figure is for scribe.

**scribemode**

Assume figure is to be included in a scribe document.

Positioning of the figure on the page will take into account 1 inch margins on all four edges of the page. The lower left corner of the figure (-1,-1) will be in the bottom of the figure area you tell scribe to use and flush against the left margin. The spacing command you give to scribe should be the same as the **-h** value given to **alps**.

**nonarrow**    Scaling is limited to the width of the page.

**narrow**       Scaling goes to full figure height specified.

Figure is tall and skinny. Normally, **alps** wants to scale your figure according to the height you give it. However, if the height is greater than the width available on the page, scaling by the full height will lose some of the edge area of the [-1,1] space. So **alps** scales only up to the available width of the page. If you know that your figure is mostly in the middle (horizontally), or if you

don't care if it falls off the sides, then setting the **narrow** flag gives you maximum scaling in the vertical direction.

**alps3D**      Use a perspective transformation.

**alps2D**      Leave out the perspective transformation.

**Alps** normally applies a perspective transformation to the input data. (2D input points are augmented to 3D with a Z value of 0.) If you need your data treated as 2D data, this flag causes the z coordinates (if present) to be ignored and no perspective to be applied. This is important for data that is truly 2D, because the implied Z=0 causes the perspective to shrink the picture.

**defprolog**    Use the default PostScript setup file.

**prolog filename**

                Use a new setup file.

A prolog file which defines PostScript functions used in the PostScript file produced by **alps** is normally included. The default file is currently "\$dispd/al\_ps\_prolog.ps". You may however, supply your own prolog file using this flag. Note that all the subroutines which are called in the data part must be defined in the prolog. You only want to do this if you are an expert at PostScript

The following set of flags are like the ones for **shape\_edit**, **view**, and other programs that display splines:

**smoothcrvs**

                Draw curves smoothly (default).

**nosmoothervs**

                Don't draw smooth curves.

**iso**

                Draw isoparametric lines on surfaces (default).

**noiso**

                Don't draw isoparametric lines on surfaces.

**nopolys**

                Don't draw curve control polygons (default).

**polys**

                Draw curve control polygons.

**nomeshes**

                Don't draw surface control meshes (default).

**meshes**

                Draw surface control meshes.

**defsrffineness**

                Use default isoparametric line spacing (50).

**srffineness float\_val**

                Set spacing for isoparametric lines.

**defcrvfineness**

                Use default curve fineness (1).

**crvfineness float\_val**

                Set curve fineness.

The **alps\_state** command provides information about the current settings of the program flags, and **alps\_defaults** resets all the flags to their default state.

**alps\_state**

*Output*

<textfile> Lists the **alps** switches that are currently set to values different than their defaults.

**alps\_defaults args**

**Output**     <Nil> Resets all the switches for **alps** to their default settings.

### Data for the Alps Program

**Alps** interprets several special attributes in the input stream which may be helpful in generating figures.

Text objects may be included in the input stream as well. These can be constructed and output easily from **shape\_edit**, but are not recognized by other system utilities (like **view** and **render**). **Alps** will map the position of the text string in the same way as other geometric data.

Several special attributes may be associated with a text\_string or placed at the top-level in the input stream to set text parameters globally. The "text\_justify" attribute has an integer value. A value of 0 indicates that the text is to be left justified at the given position, 1 indicates that it is to be centered, and 2 indicates that the text is right justified at the given position. The "font" attribute has a string value indicating the name of a font to use, and the float value of the "font\_size" attribute specifies the font size in the normal object definition space.

```
attribute( "text_justify", 0 );    % Left justify.
attribute( "text_justify", 1 );    % Center.
attribute( "text_justify", 2 );    % Right justify.

attribute( "font", "FontName" );   % Use given font.
attribute( "font_size", 0.016 );   % Size of font in object space.
```

Other attributes which are useful for controlling the appearance of a figure are "gray\_level" and the built-in "width" attribute.

```
attribute( "gray_level", 0.1 );    % Set intensity, in range [0,1]
attribute( "width", 0.02 );        % Set width of lines in object space.
```

The default width is 0.0 which produces the thinnest lines the device is able to do. Normally, lines must be very wide in order to look good with intensity values less than 0.0 (black). Default width lines may disappear completely with any other intensity value than black, depending on their slope.

You can make dashed lines using **alps** by adding a "dash" attribute to the objects before dumping them from **shape\_edit**. The attribute value is a string which is used for the dashing option in **postScript** (see that manual for full details). As an example:

```
setAttr( PLine, 'dash, "[0.1] 0" );
```

The line will be drawn with dashes 0.1 unit long with spaces 0.1 unit long. The first dash will start at the beginning and will be 0.1 unit long.

```
setAttr( QLine, 'dash, "[0.1] 0.15" );
```

This line will be drawn with dashes 0.1 unit long and will start with a gap 0.05 units long.

```
setAttr( DotDashLine, 'dash, "[0.1 0.01 0.01 0.01] 0" );
```

**DotDashLine** will be drawn with a pattern starting with a long dash (0.1 unit), a short gap (0.01 unit), a short dash, and a short gap (and then repeating).

Note that the dash dimensions are specified in **screen** space (after any viewing transformation has been applied), not in **object** space. The **alps** program attempts to verify that a dash specification is correctly formatted, but doesn't do a complete job of it.





## 18. Model Analysis

This chapter describes utilities which are available for extracting information from a completed model. This may include interfaces to finite element packages or numerically controlled machining facilities in the future. For now, the only utility available is the `calc` program which computes mass properties of a model.

### 18.1 Mass Property Calculations

This section describes the `calc` program, an Alpha\_1 utility for calculating surface area and mass properties for a surface or set of surfaces assumed to represent a closed volume.

#### Description of the Algorithm

The algorithm for calculating solid polyhedron measures (mass properties) is taken from a paper by A. M. Messner and G. Q. Taylor published in ACM Transactions on Mathematical Software, Vol. 6, No. 1, March 1980, pp. 121-130. For this discussion, the properties of concern will be mass, volume, center of gravity, and moments of inertia.

The above mass property calculations are defined as volume integrals of a given function over a closed region. The algorithm to compute these values when a boundary representation is used, is simply is to convert these volume integrals to surface integrals, using the divergence theorem, and then to approximately calculate the surface integrals using a quadrature rule.

To compute the continuous surface integral over the bounding surface, two different quadrature rules are used, depending on the input data. The first rule, used for piecewise linear surfaces, is a four point quadrature rule defined over a triangle. This means the continuous integral over a triangular region is computed as a discrete sum which is exact for polynomials up to degree three (cubic). This is both sufficient and necessary if moments of inertia are desired. Arbitrary polygons with one or more contours (concave or convex) are split into triangular components with a normal computed for each triangle (to insure that concave sections yield negative results). The calculations are then accumulated for the entire surface by summing up the calculations for all the triangles. The second rule is a 12 point rule which is exact for polynomial equations of degree 7 or less. This rule will calculate the mass properties identified above exactly for a volume bounded by bilinear surfaces. The 12 point rule is again sufficient and necessary to compute the moments of inertia exactly (up to round-off).

The two quadrature rules provide a number of strategies for computing the mass properties of a volume bounded by non-uniform rational B-splines. The first quadrature rule provides an algorithm that computes the exact mass properties for an arbitrary polyhedral volume. Surfaces may be subdivided into a polygonal representation and then the mass properties of the polyhedral volume can be calculated exactly. Empirical evidence shows that the convergence rate for the linear approximation in this case is rather slow, however. The second option is to operate on surfaces directly. The `calc` program has the ability to subdivide surfaces to an acceptable level and then to approximate them with a bilinear surface and calculate the mass properties using the second quadrature rule. This produces better results using less data. The subdivision is done depth first to eliminate high memory usage. A combination of surfaces and polygons is also acceptable, provided the set of objects bounds some volume. Such cases can arise when processing the results of a boolean combination of volumes bounded by B-splines (see chapter 14 [Combining Objects], page 191).

#### Using the Calc Program

The **calc** program recognizes **srf\_objects**, polygons, and transformations in the input stream and ignores everything else. The input stream is assumed to represent one closed volume that is bounded by the surfaces and/or polygons provided. The **calc** program follows the standard Alpha\_1 convention that the normal points into the volume that is bounded. All surfaces and/or polygons in the input must be consistently oriented. A **flip** flag exists (see below) for data that is consistently oriented the wrong way. Any transformations found in the input stream will be used to map the input objects and will affect the results. Viewing transformations will act as object transformations, if included, and will probably not produce the intended results.

The **calc** program produces textual output describing the results and the input and puts it on standard out. The number of surfaces and polygons used for the calculations are reported, along with the surface area, volume, mass, center of gravity (x,y,z), and the moments of inertia about the three axes that are calculated. A moment of inertia can also be calculated about an arbitrary axis (see below). Diagnostics are printed when degenerate (zero area) polygons are detected or degenerate objects (zero mass) are detected. Degenerate polygons are ignored, but degenerate objects cause program termination.

The **calc** commands are loaded with the "calc" keyword to **getcmds**, as in "getcmds calc". The **calcstate** command reports the values of all the switches for **calc** which have been set to values different from the defaults. The **calcddefaults** command resets all the default values.

#### **calc files**

**Output** <textfile> The calculated mass properties are reported.  
**files** <objectstream> The geometric data for which mass properties are to be computed.

#### **calcstate**

**Output** <textfile> Values of all switches are listed which are different from the defaults.

#### **calcddefaults**

**Output** <Nil> Resets all the switches for **calc** to their defaults.

The switches for **calc** are described below. The first one in each group indicates the default.

#### **nocalcflip**

**calcflip** If on, reverse the normal orientation of all the input.

**density x** Set the density of the material (default is 1).

#### **noaxis**

**axis x y z** If set, defines an arbitrary axis to use for a moment of inertia.

#### **noorigin**

**origin x y z** If set, defines the origin for the arbitrary axis. Default is (0,0,0).

If an arbitrary axis is specified, a fourth moment of inertia is computed about that axis. If only a direction is specified with the **axis** switch, then the axis is assumed to pass through the origin. If an origin is given with the **origin** switch, the axis used will pass through the given point in the given direction.

#### **Accuracy**

The **resolution** attribute that is attached to surfaces is what affects the accuracy of the calculations when surfaces are involved. This resolution is specified in absolute object space units. This means

that a surface that is 300 units long with a resolution of 1 will be subdivided until it is "flat" to within 1 unit. This applies to the `srf_mash` program and the `calc` program when surfaces are provided as input. The `srf_mash` program retains the resulting triangles on internal lists until finished, although it does the subdivision depth first, as `calc` does, which prevents very low resolutions to be used (very large memory requirements result). If the `calc` program is used to do the subdivisions, however, the intermediate surfaces are thrown out as they become unnecessary, and the program maintains a small operating size. This means calculations can be performed with the resolution set very low, although much computing time is required. It is best to start with a large resolution and decrease it as necessary.



## Appendix A. Summary of Routines

This appendix provides a summary of routines which are available in **shape\_edit**. These functions include those which are introduced in the main text of the manual, as well as some others which should not generally be necessary from the user level. Descriptions of functions which are not described in this manual may be found in the Programmer's Manual. Some of the routines in this summary are marked [old], which means that their use is no longer recommended. They are included in this summary for reference because they may occur in existing code. Whenever a routine is accessible from the graphical user interface, the menu title and menu item where it appears are noted in square brackets at the end of the description.

The routines are grouped in "packages," or sets of related routines. Each description contains the calling sequence for the routine and a very short description of the action of the routine.

### A.1 Display Control Summary

#### Basic Display Routines

- show( *Obj1*, ... )**  
Show the objects in the currently selected windows. [Object Utilities Menu: Show Object]
- unShow( *Obj1*, ... )**  
Unshow the objects from the currently selected windows. [Object Utilities Menu: Unshow Object or Unshow List]
- reShow( *Obj1*, ... )**  
Redisplay the objects. [Object Utilities Menu: Reshow Object or Reshow List]
- showPrereqs( *Obj* )**  
Display the prerequisites for a given object. [Object Utilities Menu: Show Prereqs]
- unShowPrereqs( *Obj* )**  
Unshow the prerequisites for a given object.
- showInWindow( *WindowName*, *Obj1*, ... )**  
Show the objects in the given window.
- unShowFromWindow( *WindowName*, *Obj1*, ... )**  
Unshow the objects from the given window.
- highLight( *Obj* )**  
Highlight an object on the display. [Object Utilities Menu: Highlight Object]
- showHigh( *Obj* )**  
Show an object and highlight it.
- unHighLight( *Obj* )**  
Unhighlight an object on the display. [Object Utilities Menu: UnHighlight Object]

#### Controlling Curve and Surface Display Appearance

- getCrvPolys()**  
Retrieve the current value of the **crvPolys** flag.
- setCrvPolys( *Bool* )**  
Set the current value of the **crvPolys** flag. [Screen Utilities Menu: Curve Polys]
- getSmoothCrvs()**  
Retrieve the current value of the **smoothCrvs** flag.

**setSmoothCrvs( Bool )**

Set the current value of the **smoothCrvs** flag. [Screen Utilities Menu: Smooth Curves]

**getCrvFineness()**

Retrieve the current value of the **crvFineness** flag.

**setCrvFineness( Nbr )**

Set the current value of the **crvFineness** flag. [Screen Utilities Menu: Curve Fineness]

**getSrfMeshes()**

Retrieve the current value of the **srfMeshes** flag.

**setSrfMeshes( Bool )**

Set the current value of the **srfMeshes** flag. [Screen Utilities Menu: Surface Meshes]

**getIsoLines()**

Retrieve the current value of the **isoLines** flag.

**setIsoLines( Bool )**

Set the current value of the **isoLines** flag. [Screen Utilities Menu: Isoparametric]

**getSrfFineness()**

Retrieve the current value of the **srfFineness** flag. [Screen Utilities Menu: Surface Fineness]

**setSrfFineness( Nbr )**

Set the current value of the **srfFineness** flag.

**getSrfNorms()**

Retrieve the current value of the **srfNorms** flag.

**setSrfNorms( Bool )**

Set the current value of the **srfNorms** flag. [Screen Utilities Menu: Surface Normals]

**getReverseNorms()**

Retrieve the current value of the **reverseNorms** flag.

**setReverseNorms( Bool )**

Set the current value of the **reverseNorms** flag. [Screen Utilities Menu: Reverse Normals]

**getAdjacencyVectors( SrfList )**

Get a group of vectors representing declared adjacencies for display.

**shrinkIt( Srf )**

Shrink a surface to avoid degenerate normals (for normal display).

### Controlling the Graphics Display and Windows

**grab( DeviceName )**

Set up the named device for use during this session.

**drop( DeviceName )**

Release the named device.

**interact()** Start up the graphical user interface.

**switchTo( DeviceName )**

Move the graphics displayed on the current device to a new one.

**flush( DeviceName )**

Make sure the graphics display is up to date.

**flushAll()** Make sure all the graphics displays are up to date.

**clearDev( DeviceName )**

Clear all the windows on the named device.

**newWindow( WindowName )**  
Make a new window on the current device. [Screen Utilities Menu: New Window]

**newDevWindow( DeviceName, WindowName )**  
Make a new window on the named device.

**selectWindow( WindowName )**  
Select a window on the current device. [Screen Utilities Menu: Select Window]

**selectDevWindow( DeviceName, WindowName )**  
Select a window on the named device.

**deselectWindow( WindowName )**  
Quit sending graphics commands to the named window. [Screen Utilities Menu: Deselect Window]

**deselectDevWindow( DeviceName, WindowName )**  
Quit sending graphics commands to this window on this device.

**deselectAll()**  
Deselect all the windows.

**copyDevWindow( DeviceName, WindowName )**  
Copy the named window on the named device to the current window on the current device.

**moveDevWindow( DeviceName, WindowName )**  
Same as **copyDevWindow**, but the original is removed.

**removeWindow( WindowName )**  
Remove the named window. [Screen Utilities Menu: Remove Window]

**removeDevWindow( DeviceName, WindowName )**  
Remove the window on the named device.

**clearWindow( WindowName )**  
Clear the contents of the named window. [Screen Utilities Menu: Clear Window]

**clearDevWindow( DeviceName, WindowName )**  
Clear the window on the named device.

**viewMat()** Get the current view matrix from the display.

**setWindowViewMat( WindowName, ViewMat )**  
Set the viewing matrix in a particular window.

**setDevWindowViewMat( DeviceName, WindowName, ViewMat )**  
Set the viewing matrix in a particular window on a particular device.

**objectsInWindow( WindowName )**  
Returns a copy of the group of objects in the named window.

**objectsInDevWindow( DeviceName, WindowName )**  
Returns a copy of the group of objects in window on named device.

## A.2 Sequence Summary

These routines build sequences of numbers (or points in one case) which may be useful for more advanced users in loop constructions.

**append( List1, List2 )**  
Append two lists to form a single list.

**list( *Item1*, *Item2*, ..., *ItemN* )**

Form a list of items.

**first( *List* )**

Get the first item in a list.

**second( *List* )**

Get the second item in a list.

**third( *List* )**

Get the third item in a list.

**fourth( *List* )**

Get the fourth item in a list.

**nth( *List*, *Index* )**

Get the nth item in a list.

**seq0( *Last* )**

Returns a list of the integers 0 through *Last*.

**seq1( *Last* )**

Returns a list of the integers 1 through *Last*.

**numRamp( *NItems*, *Num1*, *Num2* )**

Returns a list of floating numbers interpolated between *Num1* and *Num2*.

**numStep( *N*, *StartVal*, *StepSize* )**

Make a list of *N* numbers given starting value and step size.

**nTimes( *NItems*, *Item* )**

Returns a list containing *NItems* copies of *Item*.

**ptsOnLineList( *P1*, *P2*, *NumPts* )**

Returns a list of points spaced evenly along the line between *P1* and *P2*.

## A.3 Points Summary

### Constructors

**pt( *x*, *y* )**

**pt( *x*, *y*, *z* )**

Returns a point with the given coordinates. [Points Menu: Coordinates]

**projPt( *x*, *y*, *w* )**

**projPt( *x*, *y*, *z*, *w* )**

Returns a point with the given coordinates. [Points Menu: Projective By Coords]

**vec( *a*, *b*, ... )**

Returns a vector with the given coordinates. [Vectors Menu: Coordinates]

**vecOrigin( *Dim* )**

Returns a vector of given dimension with all coordinates 0.

### Extractors

**ptDim( *Pt* )**

Returns dimension (2 or 3) of a point.

**vecDim( *Vec* )**

Returns dimension of a vector.



**ptSize( Pt )** Returns number of coordinates (2, 3, or 4) of a point.

**vecSize( Vec )** Same as **vecDim**.

**ptX( Pt )** Returns X coordinate of a point. [Numbers Menu: X Coordinate]

**ptY( Pt )** Returns Y coordinate of a point. [Numbers Menu: Y Coordinate]

**ptZ( Pt )** Returns Z coordinate of a point. [Numbers Menu: Z Coordinate]

**ptW( Pt )** Returns W coordinate of a point. [Numbers Menu: W Coordinate]

**vecX( Vec )** Returns X coordinate of a vector. [Numbers Menu: X Coordinate]

**vecY( Vec )** Returns Y coordinate of a point. [Numbers Menu: Y Coordinate]

**vecZ( Vec )** Returns Z coordinate of a vector. [Numbers Menu: Z Coordinate]

#### Coercions

**e2( Pt )** Coerce point to euclidean 2D point.

**p2( Pt )** Coerce point to projective 2D point.

**e3( Pt )** Coerce point to euclidean 3D point.

**p3( Pt )** Coerce point to projective 3D point.

**r2( Vec )** Coerce vector to 2D.

**r3( Vec )** Coerce vector to 3D.

#### Predicates

**pointP( Pt )** Returns true value only if argument is a point.

**projectiveP( Pt )** Returns true value only if argument is a projective point.

**euclideanP( Pt )** Returns true value only if argument is a euclidean point.

**vecP( Vec )** Returns true value only if argument is a (geometric) vector.

#### Operations on Points and Vectors

**vecAtAngle( Angle )** Returns a unit R2 vector at a given angle. [Vectors Menu: At Angle]

**vecAngle( Vec )** Returns angle of vector, in degrees CCW from 0 to the right (+X).

**vecLength( Vec )** Returns the length of a vector. [Numbers Menu: Vector Length]

**unitVec( Vec )** Normalizes a vector. [Vectors Menu: Unit Vector]

**vecPlus( Vec1, Vec2 )** Computes the sum of two vectors. [Vectors Menu: Add Vectors]

**vecMinus( Vec1, Vec2 )** Computes the difference of two vectors. [Vectors Menu: Subtract Vectors]

**ptOffset( Pt, Vec )**

Computes the point offset from the given point by the vector. [Points Menu: Offset]

**ptScaledOffset( Pt, Vec, Factor )**

Like **ptOffset**, but vector is scaled before offset is done. [Points Menu: Scaled Offset]

**vecOffset( BasePt, EndPt )**

Computes vector between two given points. [Vectors Menu: 2 Points]

**vecFrom2Pts( BasePt, EndPt )**

Alternate name for **vecOffset**. [Vectors Menu: 2 Points]

**ptMinus( Pt1, Pt2 )**

Computes vector between two given points, reverse order of arguments.

**vecScale( Vec, Factor )**

Scales a vector. [Vectors Menu: Scale]

**distPtPt( Pt1, Pt2 )**

Computes the distance between two points. [Numbers Menu: 2 Pt Distance]

**dotProd( Vec1, Vec2 )**

Computes the dot product of two vectors.

**crossProd( Vec1, Vec2 )**

Computes the cross product of two (r2 or r3) vectors. [Vectors Menu: 2 Vector Perp]

#### Blending Functions

**ptInterp( Pt1, Pt2, Parm2 )**

Computes the point on the line through the two given points at the specified parametric value. [Points Menu: 2 Pt Interp]

**vecInterp( Vec1, Vec2, Parm2 )**

Computes the vector which is the linear interpolation of the two given vectors at the specified parametric value. [Vectors Menu: 2 Vec Interp]

**ptBlend( PtList, CoeffList )**

Convex blending of a list of euclidean points.

**vecBlend( VecList, CoeffList )**

Convex blending of a list of vectors.

## A.4 Lines Summary

**linePtAngle( Pt, Angle )**

Constructs a line through the point at the given angle. [Lines & Planes Menu: Point & Angle]

**linePtVec( Pt, Vec )**

Constructs a line through the point in the given vector direction. [Lines & Planes Menu: Point & Direction]

**lineThru2Pts( Pt1, Pt2 )**

Constructs a line through the given points. [Lines & Planes Menu: 2 Points]

**linePtParallel( Pt, Ln )**

Constructs a line through a point parallel to the given line. [Lines Menu: Point Parallel]

**lineOffsetFromLine( Offset, Ln )**

Constructs a line offset from the given line by the Offset. [Lines Menu: Offset Line]

- linePtCircle( *Pt*, *Cir*, *CcwFlag* )**  
Constructs a line from a point tangent to a circle. [Lines & Planes Menu: Point & Circle]
- lineTan2Circles( *Cir1*, *Ccw1*, *Cir2*, *Ccw2* )**  
Constructs a line tangent to two circles. [Lines & Planes Menu: 2 Circles]
- lineVertical( *Xoffset* )**  
Constructs a vertical line at the given X coordinate. [Lines & Planes Menu: Vertical]
- lineHorizontal( *Yoffset* )**  
Constructs a horizontal line at the given Y coordinate. [Lines & Planes Menu: Horizontal]
- lineIntersect2Planes( *Plane1*, *Plane2* )**  
Computes the line of intersection of two planes.
- reverseLine( *Ln* )**  
Reverses the orientation of a line. [old]
- normalizeLine( *Line* )**  
Normalize line coordinates.
- ptIntersect2Lines( *Line1*, *Line2* )**  
Computes the point at the intersection of the given lines. [Points Menu: 2 Lines Intersect]
- ptOnLineNearestLine( *Line1*, *Line2* )**  
Computes the point on one line nearest another line.
- ptNearest2Lines( *Line1*, *Line2* )**  
Computes the point nearest two lines.
- segBetween2Lines( *Line1*, *Line2* )**  
Computes the shortest line segment joining two lines.
- distLineLine( *Line1*, *Line2* )**  
Computes the minimum distance between two lines.
- projectPtOntoLine( *Pt*, *Ln* )**  
Perpendicular projection of the point onto the line. [Points Menu: Pt Project to Line]
- distPtLine( *Pt*, *Ln* )**  
Computes the (positive) distance from the point to the line. [Numbers Menu: Pt Line Distance]
- signedDistPtLine( *Pt*, *Ln* )**
- signedDistPtLine( *Pt*, *Ln*, *ReferenceVec* )**  
Computes the signed distance from the point to the line. *ReferenceVec* is an optional argument for governing the sign.
- dirOfLine( *Ln* )**  
Returns a vector in the direction of the line. [Vectors Menu: Line Direction]
- dirPerpLine( *Ln* )**  
Returns a vector in the direction of the perpendicular to an e2 line. [Vectors Menu: Perp Line Direction]
- dirPerpLineInPlane( *Ln*, *Plane* )**  
Returns a vector parallel to a plane and perpendicular to a line.
- angleOfLine( *Ln* )**  
Returns the angle of the direction of a line, degrees CCW from +X.

**ptOnLine( Ln )**

Returns a euclidean point on a line.

**ptOnLineWithX( Line, XCoord )**

**ptOnLineWithY( Line, YCoord )**

**ptOnLineWithZ( Line, ZCoord )**

Return a point on a line with a specified coordinate value.

**e2( Line )** Coerce line to 2D line.

**e3( Line )** Coerce line to 3D line.

**lineP( Any )**

Tests whether an object is a line.

**linesParallelP( Line1, Line2 )**

Tests whether two lines are parallel.

**linesSkewP( Line1, Line2 )**

Tests whether two lines are skew.

### **Polylines and Polygons**

**polyline( PtList )**

Create a polyline with the given points. [Points Menu: Polyline]

**newPolyline( Sz, PtType )**

Create new polyline of given size.

**polylineSize( PolyL )**

Returns size of the polyline.

**reversePolyline( PolyL )**

Reverse the order of the points. [old]

**polygon( PtList )**

Create polygon with given points (implicitly closed). [Points Menu: Polygon]

**newPolygon( Sz, PtType )**

Create new polygon of given size.

**polygonSize( Poly )**

Returns size of the polygon.

**reversePolygon( Poly )**

Reverse the order of the points. [old]

**polylineFromPolygon( Poly )**

Return a polyline representing the same polygon (explicitly closed).

## **A.5 Planes Summary**

**normalizePlane( Plane )**

Normalizes the plane equation.

**planeOffsetByDelta( Plane, Delta )**

Constructs a plane offset from another plane. [Lines & Planes Menu: Offset Plane]

**planeThruPtAndLine( Point, Line )**

Constructs a plane through a point and line.

**planeThru3Pts( Pt1, Pt2, Pt3 )**

Constructs a plane through three points.

- planeThru2Lines( *Line1*, *Line2* )**  
 Constructs a plane containing two (non-skew) lines.
- planeThruLineParallelToLine( *Line1*, *Line2* )**  
 Constructs a plane containing the first line and parallel to the second.
- reversePlane( *Plane* )**  
 Reverses the orientation of a plane. [old]
- planeThruPtWithNormal( *Point*, *NormalDir* )**  
 Constructs a plane through the given point with specified normal. [Lines & Planes Menu: Point & Normal]
- planeNormal( *Plane* )**  
 Returns a vector which is the normal to the plane. [Vectors Menu: Plane Normal]
- signedDistPtPlane( *Point*, *Plane* )**  
 Returns the signed perpendicular distance from the point to the plane. [Numbers Menu: Pt Plane Distance]
- angleFrom2Planes( *Plane1*, *Plane2* )**  
 Computes the angle between two planes.
- ptFrom3Planes( *Plane1*, *Plane2*, *Plane3* )**  
 Computes the point at the intersection of three planes. [Points Menu: 3 Planes]
- ptIntersectLineAndPlane( *Line*, *Plane* )**  
 Computes the point of intersection of a line and plane.
- vecProjVecOntoPlane( *Vec*, *Plane* )**  
 Returns a vector which is the projection of the given vector onto the plane. [Vectors Menu: Project Vec to Plane]
- ptProjPtDirPlane( *Point*, *Direction*, *Plane* )**  
 Returns a point which is the projection of the given point onto the plane. [Points Menu: Pt Project to Plane]
- crvFromCrvProjOntoPlane( *Curve*, *Projector*, *Plane* )**  
 Returns the projection of a curve onto the plane. [Basic Curves Menu: Project Crv To Plane]

## A.6 Arcs & Circles Summary

- arcTan3Lines( *Line1*, *Line2*, *Line3* )**  
 Constructs the arc which is tangent to the three given lines. [Arcs & Circles Menu: 2 Lines]
- arcRadTan2Lines( *Radius*, *Line1*, *Line2* )**  
 Constructs the arc of given radius tangent to the two lines. [Arcs & Circles Menu: Radius & 2 Lines]
- arcThru3Pts( *EndPt1*, *MiddlePt*, *EndPt2* )**  
 Constructs the arc which passes through the three points. [Arcs & Circles Menu: 3 Points]
- arcEndCenterEnd( *EndPt1*, *CenterPt*, *EndPt2* )**  
 Constructs the arc which has the given endpoint and center. [Arcs & Circles Menu: End Center End]
- arcEndTan2Lines( *EndPt1*, *Line1*, *Line2*, *OtherSide* )**  
 Constructs an arc starting at a point and tangent to two specified lines.

**reverseArc( Ar )**  
Reverse the orientation of an arc. [old]

**arcRadTanToCircleAndLine( Radius, CircleCtr, CircleRad, TanLine )**  
Construct an arc which is tangent to a circle (defined by its center and radius) and a line. [Arcs & Circles Menu: Radius Circle Line]

**radiusOfArc( Ar )**  
Compute the radius of the given arc. [Numbers Menu: Radius]

**centerOfArc( Ar )**  
Compute the center point of the given arc. [Points Menu: Center Of]

**arcStart( Arc )**  
Return the first endpoint of the arc. [Points Menu: Arc Start]

**arcEnd( Arc )**  
Return the last endpoint of the arc. [Points Menu: Arc End]

**endDirOfArc( Ar )**  
Compute tangent vector at end of arc.

**startDirOfArc( Ar )**  
Compute tangent vector at beginning of arc.

**directionOfArc( Ar )**  
Return 'CW or 'CCW for direction of the arc.

**otherArc( Arc )**  
Constructs an arc representing the other part of the circle. [Arcs & Circles Menu: Other Arc]

**circleCtrRad( Ctr, Rad )**  
Construct circle with given center and radius. [Arcs & Circles Menu: Center Radius]

**circleCtrPt( Ctr, Edge )**  
Construct circle with given center passing through point. [Arcs & Circles Menu: Center Point]

**circleCtrRadNormal( Ctr, Rad, Normal )**  
Construct a circle with given center and radius, in plane specified by normal.

**circleRad( Circle )**  
Extract radius from a circle. [Numbers Menu: Radius]

**circleCtr( Circle )**  
Extract center from a circle. [Points Menu: Center Of]

**arcCutFromCircle( Cir, Line )**  
Cut an arc from the circle by passing a line through. [Arcs & Circles Menu: Cut From Circle]

**circleRadTan2Circles( Rad, Cir1, CSW1, Cir2, CSW2 )**  
Construct a circle with given radius, tangent to two other circles. [Arcs & Circles Menu: Radius 2 Circles]

**arcRadTan2Circles( Rad, Cir1, CSW1, Cir2, CSW2 )**  
Construct an arc with given radius, tangent to two other circles. [Arcs & Circles Menu: Radius 2 Circles]

**ptIntersect2Circles( Cir1, Cir2 )**  
Calculate one of the intersection points of two circles. [Points Menu: 2 Circles Intersect]

**ptIntersectCircleLine( Cir, Line )**  
Construct a point at the intersection of a circle and a line.

## A.7 Control Mesh Summary

- ctlPoly( *Elements* )**  
Construct and fill in new control polygon from list or vector.
- newCPolygon( *Sz*, *PtType* )**  
Constructs a new control polygon for a curve.
- cPolySize( *Cp* )**  
Returns the size of a control polygon.
- ptListFromCtlPoly( *CtlPoly* )**  
Return a list of the control points in the polygon.
- ctlMesh( *Elements* )**  
Construct and fill in new control mesh from lists or vectors. [Points Menu: Point Mesh]
- newCMesh( *Rsize*, *Csize*, *PtType* )**  
Constructs a new control mesh for a surface.
- cMeshSize( *Mesh*, *Dir* )**  
Returns the size of the control mesh in the given direction.
- ctlMeshTranspose( *Mesh* )**  
Transpose the elements of a control mesh.
- reverseMeshInDir( *Mesh*, *Dir* )**  
Reverse the rows or columns of a control mesh.

## A.8 Knot Vector Summary

- knotVector( *Elements* )**  
Construct and fill in new knot vector from list or vector. [Parminfo Menu: Number List]
- newKnotVector( *Sz* )**  
Constructs a new knot vector of the given size.
- kvSize( *Kv* )**  
Returns the size of the knot vector.
- reverseKv( *Kv* )**  
Reverse the knot vector.
- parmInfo( *Order*, *EcType*, *Kv* )**  
Constructs a parminfo for use in curves or surfaces. [Parminfo Menu: Create Parminfo]
- parmOrder( *ParmInfo* )**  
Returns the order of the parminfo.
- parmEndCondType( *ParmInfo* )**  
Returns the end condition type of the parminfo.
- parmKvType( *ParmInfo* )**  
Returns the knot vector type of the parminfo.
- parmKvValue( *ParmInfo* )**  
Returns the values in the knot vector, if they exist.
- kvNormalize( *Kv* )**  
Returns an equivalent knot vector which begins at value 0 and ends at value 1. [3]
- mergeKv( *OrigKnots*, *NewKnots* )**

Return a new merged knot vector.

## A.9 Curves Summary

- crvFromArc( Arc )**  
Converts an arc into a spline curve. [Basic Curves Menu: From Arc]
- crvFromCircle( Circ )**  
Converts a circle into a spline curve. [Basic Curves Menu: From Circle]
- curve( Parms, CtlPoly )**  
Constructs a curve with the given parmInfo and control polygon. [Basic Curves Menu: Curve]
- cParmInfo( Crv )**  
Returns the parmInfo of the curve. [ParmInfo Menu: From Curve]
- cOrder( Crv )**  
Returns the order of the curve.
- cType( Crv )**  
Returns the end condition type of the curve.
- cKvType( Crv )**  
Returns the type of the associated knot vector.
- cKv( Crv )**  
Returns the value of the associated knot vector.
- cPoly( Crv )**  
Returns the control polygon of the curve.
- crvPt( Crv, Indx )**  
Extract a single control point of a curve.
- profile( Thing1, Thing2, ... )**  
Joins a list of curves, arcs, or points into a single curve. [Basic Curves Menu: Profile]
- reflect( Crv, Axis )**  
Reflect a curve by negating the specified coordinates. [Points Menu: Reflect Point] or [Basic Curves Menu: Reflect]
- cRefine( OldCrv, NewKnots, DoMerge )**  
Refine the curve with the given knots. [Basic Curves Menu: Refine]
- raiseCurve( Crv, DesiredOrder )**  
Raise the order of the curve to the given order.
- raiseOrder( Obj, DesiredOrder )**  
Raise the order of any curves or surfaces in *Obj* to given order.
- raiseCrvOrder( Obj, DesiredOrder )**  
Raise the order of curves (only) in *Obj* to given order.
- crvEval( Crv, Tee )**  
Evaluate the curve at the given parametric value. [Points Menu: Curve Eval]
- diffCrv( Crv, N )**  
Compute the curve which is the Nth derivative of the given curve.
- crvDerivEval( Crv, Tee )**  
Evaluate the derivative of the curve at the given parametric value. [Vectors Menu: Curve Derivative]



**crvNthDerivEval( Crv, N, Tee )**  
 Evaluate the n-th derivative of the curve at the given parametric value. [Vectors Menu: Curve N Derivative]

**reverseCrv( Crv )**  
 Reverses the orientation of things that might be curves, including curves, arcs, and lists of points. [old]

**mkCompatible( Crv1, Crv2, ..., PtCoerceFlag )**  
 Returns a list of equivalent curves which all have the same order and knot vectors.

**sameOrder( Crv1, Crv2, ... )**  
 Returns a list of equivalent curves which all have the same order.

**curveOpen( Curve )**  
 Convert curve to open end conditions. [Basic Curves Menu: Open End Condition]

**regionFromCrv( Crv, MinVal, MaxVal )**  
 Return a curve representing part of the original. [Basic Curves Menu: Curve Region]

**computeNodes( CrvOrParmInfo )**  
 Compute the nodes for a curve or parmInfo.

## A.10 Surface Summary

**surface( Parms, Mesh )**  
 Constructs a surface from the given parminfos and control mesh. [Basic Surfaces Menu: Surface]

**srfFromCrvs( Parminfo, Crv1, Crv2, ... )**  
 Constructs a surface with rows given by the curves and the specified parmInfo used in the column direction.

**srfFromCrvsDir( Parminfo, Dir, Crv1, Crv2, ... )**  
 Constructs a surface with the curves making up the control mesh running in the specified direction and the specified parmInfo in the other direction.

**sParmInfos( Srf )**  
 Returns the parmInfos of the surface.

**rowParmInfo( Srf )**  
 Get the parmInfo in the row direction. [ParmInfo Menu: From Surface U (ROW) Dir]

**colParmInfo( Srf )**  
 Get the parmInfo in the column direction. [ParmInfo Menu: From Surface V (COL) Dir]

**sOrders( Srf )**  
 Returns the order of the surface (one order for each direction).

**sTypes( Srf )**  
 Returns the end condition types for the surface.

**sKvTypes( Srf )**  
 Returns the knot vector types of the surface.

**sKvs( Srf )**  
 Returns the values of the surface knot vectors.

**sMesh( Srf )**  
 Returns the control mesh of the surface.

- reverseSrfInDir( Srf, Dir )**  
Reverses the isoparametric curves of the surface in the given direction.
- reverseSrf( Srf )**  
Reverses the orientation of the surface. [old]
- crvFromSrf( Srf, Dir, Index )**  
Extract the specified row or column of the surface control mesh as a curve. (Curve does not necessarily lie in the surface). [Basic Curves Menu: From Surface Mesh]
- crvInSrf( Srf, Dir, ParamVal )**  
Extract the given isoparametric curve of the surface. [Basic Curves Menu: In Surface]
- getBoundary( Srf, Bndary )**  
Extract one of the four surface boundaries as a curve. [Basic Curves Menu: Surface Boundary]
- sRefine( OldSrf, Dir, NwKnots, DoMerge )**  
Refine the surface in the specified direction with the given set of knots. [Basic Surfaces Menu: Refine]
- srfDivide( Surface, Dir )**  
Return a list of two surfaces formed by subdividing in given direction.
- srfSubdiv( Surface, Dir, MidIndex, SplitMult, KnotVal )**  
Like **srfDivide**, only you get to say where to subdivide.
- regionFromSrf( Surface, Dir, LowerParam, UpperParam )**  
Extract region between lower and upper parameters in given direction. [Basic Surfaces Menu: Surface Region]
- surfaceOpen( Surface )**  
Convert surface to open end conditions. [Basic Surfaces Menu: Open End Condition]
- srfMerge( Srf1, Srf2, Dir, JoinType )**  
Merges two surfaces together along the given boundary.
- surfaceNormals( S, Uvals, Vvals )**  
Calculates the surface normals of S at the cartesian product of the U and V parametric values given.
- srfEval( Srf, U, V )**  
Evaluate the surface for the given U and V value.
- diffSrf( Srf, UNum, VNum )**  
Compute the partial derivative of a surface. *UNum* and *VNum* specify how many times the surface is differentiated in the parametric directions.
- raiseSurface( Srf, Dir, Order )**  
Raise the order of the surface in the given direction to the specified order.
- raiseOrder( Obj, Order )**  
Raise the order of any curves or surfaces in *Obj* to the given order. Both directions of surfaces are raised. [Basic Curves Menu: Raise Degree]
- raiseSrfOrder( Obj, Dir, Order )**  
Raise the order in the given direction of surfaces (only) in *obj*.
- srfAndDerivs( Srf )**  
Calculate derivative information for a surface and cache it.
- srfGeom( SrfStruct, U, V )**  
Calculate surface geometry (curvature, etc.) at given point.
- prinCircles( SrfGeom )**

Return a group of osculating circles determined by **srfGeom**.

**onePrinCircle**( *Position, Normal, PrinDir, PrinCurv* )

Create an osculating circle given relevant geometric information.

## A.11 Primitive Summary

**box**( *Vertex, Height, Width, Depth* )

Returns a box with corner at given *Vertex* and specified *Height, Width* and *Depth*. Geometry field has a shell to represent the box. [Primitives Menu: Box With Vecs]

**rightAngleWedge**( *Vertex, Height, Width, Depth* )

Returns a wedge object. Geometry field has a shell to represent the wedge. [Primitives Menu: Wedge]

**rightCirCylinder**( *Vertex, Height, Radius* )

Returns a cylinder object. Geometry field has a shell to represent the cylinder. [Primitives Menu: Cylinder]

**truncRightCone**( *Vertex, Height, Radius1, Radius2* )

Returns a truncated right cone. Geometry field has a shell to represent the cone. [Primitives Menu: Truncated Cone]

**sphere**( *Vertex, Radius* )

Returns a sphere object. Geometry field has a shell to represent the sphere. [Primitives Menu: Sphere]

**ellipsoid**( *Vertex, AxisOfRev, Radius* )

Returns an ellipsoid object. Geometry field has a shell to represent the ellipsoid.

**torus**( *Vertex, PlaneNormal, BigRadius, SmallRadius* )

Returns a torus object. Geometry field has a shell to represent the torus. [Primitives Menu: Torus]

**roundRightCirCylinder**( *Vertex, Height, Radius, Fillet1, Fillet2* )

Returns a rounded cylinder. Geometry field has a shell to represent the round cylinder. [Primitives Menu: Rounded Cylinder]

**roundTruncCirCone**( *Vertex, Height, Rad1, Rad2, Fillet1, Fillet2* )

Returns a rounded cone. Geometry field has a shell to represent the round cone. [Primitives Menu: Rounded Truncated Cone]

**rboxFrom6Planes**( *Right, Left, Up, Down, Front, Back* )

Build a rounded box from 6 planes. Initially, none of the edges are rounded. [Primitives Menu: Box With Planes or Rounded Box]

**rboxOffsetFromRbox**( *Rbox, Offset* )

Construct a rounded box which is an offset from an existing one. [Primitives Menu: Offset Rbox]

**setRboxFacesOpen**( *Rbox, Faces* )

Set the named faces of the given rounded box to be open. [Primitives Menu: Set Rbox Faces Open]

**setRboxRadius**( *Rbox, Edge, RadiusValue* )

Set the radius on the named edge. [Primitives Menu: Set Rbox Radius]

## A.12 Surface Operators Summary

**bend( *SList*, *Axis*, *BendAngle*, *BendCenter*, *MinRange*, *MaxRange*, *LH* )**

Bend a surface or list of surfaces relative to an axis. [Shape Operations Menu: Bend]

**boolSum( *CrvU0*, *CrvU1*, *Crv0V*, *Crv1V* )**

Construct the bilinearly blended boolean sum surface with the four given boundary curves. [Basic Surfaces Menu: 4 Boundary Curves]

**extrude( *Crv*, *Pt1*, *Pt2* )**

Construct a surface by extruding the curve, moving origin to *Pt1* and *Pt2*. [Basic Surfaces Menu: 2 Pt Extrude]

**extrudeDir( *Crv*, *DirVector* )**

Construct a surface by extruding the curve in the given direction. [Basic Surfaces Menu: Direction Extrude]

**flatSrfBounds( *Uorder*, *Vorder*, *Vsize*, *Usize*, *MinPt*, *MaxPt* )**

Construct a flat surface in the XY-plane with the given bounds. [Basic Surfaces Menu: Bounded Flat]

**flatSrf( *Uorder*, *Vorder*, *Vsize*, *Usize* )**

Construct a flat surface in the XY-plane extending from -1 to 1. [Basic Surfaces Menu: Flat]

**flattenSrf( *SList*, *P*, *Dir*, *OtherSide* )**

Flatten a surface or list of surfaces with a plane. [Shape Operations Menu: Flatten]

**flattenSrfR( *SList*, *P*, *Dir*, *OtherSide*, *RestrictList* )**

Like **flattenSrf**, but the effect is limited by the restriction list.

**addFlex( *S*, *Axis*, *MinVal*, *MaxVal*, *Number*, *RefCrv* )**

Add flexibility (extra control points) to the given surface. [Shape Operations Menu: Add Flex]

**addFlexObj( *Obj*, *Axis*, *MinVal*, *MaxVal*, *Number*, *RefCrv* )**

Like **addFlex**, but applies it to all the surfaces in *Obj*.

**isolateRegion( *S*, *Axis*, *MinVal*, *MaxVal*, *RefCrv* )**

Isolate the specified region of the surface so that changes within do not propagate outside. [Shape Operations Menu: Isolate Region]

**featureLine( *S*, *Axis*, *Value*, *ContinuityClass*, *RefCrv* )**

Add a potential line of discontinuity to the surface. [Shape Operations Menu: Feature Line]

**lift( *SList*, *Direction*, *DistInfo* )**

Perform lifting of a surface or list of surfaces. [Shape Operations Menu: Lift]

**loftHit( *S*, *Dir*, *GeomFn*, *ExtraArgs* )**

Apply given function to all the curves in a surface.

**singleSrfEdge( *TopSrf*, *BotSrf*, *WhichEdge*, *CrossSectionType*, *CrossSectionInfo* )**

Construct a surface joining one edge of each of the two given surfaces. [Shape Operations Menu: Join Linear or Join Quadratic or Join Cubic]

**reflect( *SList*, *Axis* )**

Reflect a surface or list of surfaces about the given axis.

**regionWarp( *SList*, *DirVec*, *WarpFactor*, *Cutoff*, *MeasureInfo*, *Region* )**

Warp a surface or list of surfaces based on a given region. [Shape Operations Menu: Region Warp]

**regionWarpR( *SList*, *DirVec*, *WFactor*, *Cutoff*, *MInfo*, *Region*, *RestrictList* )**

Like **regionWarp**, but effect is limited by the restriction list.

- buildRestriction( *RestrictionList* )**  
Constructs a list of restrictions to be used in **flattenSrf**.
- addRestriction( *OldRestrictList*, *NewRestrictions* )**  
Add some more restrictions to an existing restriction list.
- checkRestrictions( *Pt*, *RestrictionList* )**  
Determine if a point is excluded by the given restriction list.
- ruledSrf( *Crv1*, *Crv2* )**  
Construct a ruled surface from two curves. [Basic Surfaces Menu: Ruled]
- circTubeConstantWidth( *Axis*, *Radius* )**  
Creates a simple sweep of constant width and circular cross section.
- circTubeWithProfile( *Axis*, *Profile* )**  
Creates a sweep of variable width and circular cross section.
- sweepConstantWidth( *Axis*, *CrossSec*, *Scale*, *XInNormPlane* )**  
Creates a sweep of constant width and arbitrary cross section.
- sweepWithProfile( *Axis*, *CrossSec*, *Profile*, *XInNormPlane* )**  
Creates a sweep of variable width and arbitrary cross section.
- generalSweep( *Axis*, *CrossSec*, *BlendFlag*, *Profile*, *XInNormPlane* )**  
Creates a sweep using blending of cross sections and/or variable scaling.
- vOffset( *Srf*, *DistInfo* )**  
Construct a surface offset from another surface.
- warp( *Obj*, *Direction*, *Center*, *WFactor*, *Cutoff*, *MInfo* )** Construct a surface by "warping" another surface.
- warpR( *Obj*, *Direction*, *Center*, *WFactor*, *Cutoff*, *MInfo*, *Restrict* )**  
A variant of **warp** which constructs a warped surface with the warp restricted to certain regions.
- skelWarp( *SList*, *DirVec*, *WarpFactor*, *Cutoff*, *MeasureInfo*, *Skel* )**  
Warp a surface or list of surfaces based on the given skeleton. [Shape Operations Menu: Skeletal Warp]
- skelWarpR( *SList*, *DirVec*, *WFactor*, *Cutoff*, *MInfo*, *Skel*, *RestrictList* )**  
Like **skelWarp**, but the effect is limited by the restriction list.
- srfEdge( *TopSrf*, *BotSrf*, *CrossSectionType*, *CrossSectionInfo* )**  
Construct an edge surface joining the boundaries of the two given surfaces. [Shape Operations Menu: Thicken]
- srfOfRevolution( *AxisLine*, *ProfileCrv* )**  
Construct a surface of revolution given the axis and the curve to be rotated. [Basic Surfaces Menu: Of Revolution]
- shellOfRevolution( *AxisLine*, *ProfileCrv* )**  
Same as **srfOfRevolution**, except that a shell structure is formed.
- srfAxisProfileSection( *AxisLine*, *ProfileCrv*, *SectionCrv* )**  
More general than **srfOfRevolution**, this one allows arbitrary cross section shape. [Basic Surfaces Menu: Profile & Section]
- shellAxisProfileSection( *AxisLine*, *ProfileCrv*, *SectionCrv*, *EndCap* )**  
As above, but returns a shell structure.
- stretch( *SList*, *XScale*, *YScale*, *ZScale* )**  
Scale a surface or list of surfaces in the axial directions. [Shape Operations Menu: Stretch]

- taper( *SList, Axis, TaperFn, LH* )**  
Scale a surface or list of surfaces along the given axis with scaling values determined by the tapering routine.
- twist( *SList, Axis, TwistFn* )**  
Rotate a surface or list of surfaces about the given axis, with rotation angle determined by the twisting routine.
- taperWithSplineFn( *Obj, Axis, LH, TaperFn* )**  
Like **taper**, but uses a spline curve function to control shape. [Shape Operations Menu: Taper]
- twistWithSplineFn( *Obj, Axis, TwistFn* )**  
Like **twist**, but uses a spline curve function to control shape. [Shape Operations Menu: Twist]
- constantSplineFn( *BeginParm, EndParm, Val* )**  
Construct a constant spline curve for use as a function  $f(t)$ . [Spline Functions Menu: Constant]
- linearSplineFn( *BeginParm, EndParm, BeginVal, EndVal* )**  
Construct a linear spline curve for use as a function  $f(t)$ . [Spline Functions Menu: Linear]
- approxValuesSplineFn( *BeginParm, EndParm, ValList, UseY* )**  
Construct a spline curve for use as a function  $f(t)$  by approximating a set of values. [Spline Functions Menu: Approximate Values]
- interpValuesSplineFn( *BeginParm, EndParm, ValList, FnY* )**  
Construct a spline curve for use as a function  $f(t)$  by interpolating a set of values. [Spline Functions Menu: Interpolate Values]

### A.13 Curve Operators Summary

- crvTangLines( *LineList* )**  
Construct a Bezier style curve which has tangencies according to the line and intersection specifications given. [Basic Curves Menu: Tangent Lines]
- crvTangLines2( *LineList* )**  
Construct a curve which has tangencies according to the line and intersection specifications given.
- cAddFlex( *C, Axis, MinVal, MaxVal, Number* )**  
Add flexibility (additional control points) to a curve.
- cIsolateRegion( *C, Axis, MinVal, MaxVal* )**  
Isolate a region of a curve.
- cFeatureLine( *C, Axis, Value, ContinuityClass* )**  
Add a potential discontinuity to a curve.
- cLift( *CList, Direction, DistInfo* )**  
Perform lifting of a curve or list of curves.
- qOffset( *Crv, RefVec* )**  
Compute an offset curve. [Basic Curves Menu: Offset]
- cvOffset( *CList, DistInfo* )**  
Offset a curve or list of curves according to the given offset distance information.

## A.14 Interpolation Summary

**completeCubicInterp**( *Parameters*, *Positions*, *BeginTangent*, *EndTangent* )

Return a curve interpolating the given data. [Curve Fitting Menu: Complete Cubic]

**cubicCrvFromParametersAndPositions**( *Parameters*, *Positions* )

Return a curve interpolating the given data. [Curve Fitting Menu: Cubic]

**crvFromParmInfoAndPositions**( *ParmInfo*, *Positions* )

Return a curve interpolating the given data. [Curve Fitting Menu: Using Parminfo]

**periodicCompleteCubicInterp**( *Parms*, *Posns*, *ParamEstArg1*, ..., *ParamEstArgN* )

Return a curve interpolating the given data. [Curve Fitting Menu: Periodic Complete Cubic]

## A.15 Aggregate Summary

**group**( *Obj1*, *Obj2*, ... )

Form a group from the given objects. [Aggregates Menu: Group]

**addToGroup**( *Group*, *Obj* )

Add an object to a group. [Aggregates Menu: Add To Group]

**deleteFromGroup**( *Group*, *Obj* )

Delete an object from a group. [Aggregates Menu: Delete From Group]

**memberOfGroup**( *Group*, *Obj* )

Find out if an object is in the group.

**instance**( *Obj*, *Trans1*, *Trans2*, ... )

Instance the given object according to the transformation sequence. [Aggregates Menu: Instance]

**replaceObject**( *Instance*, *Obj* )

Replace the object currently in the given instance with a new one. [Aggregates Menu: Replace Instance]

**libraryEntry**( *Obj* )

Tag an object as a library entry.

**shell**( *SrfList* )

Form a shell from the given surface list. [Aggregates Menu: Shell]

**combinerObject**( *Expression* )

Make a combiner object which represents the expression.

## A.16 Transformation Summary

**objTransform**( *Obj*, *Trans1*, *Trans2*, ... )

Map the given object through the transformation sequence.

**invObjTransform**( *Obj*, *Trans1*, *Trans2*, ... )

Map the object through the inverse of the transformation sequence.

**transform**( *Descr1*, ... )

Concatenate a sequence of matrix descriptors into a single transform.

**rotateX**( *Angle* )

Rotation about X axis by given angle. [Transformations Menu: Rotate]

**rotateY( Angle )**  
 Rotation about X axis by given angle. [Transformations Menu: Rotate]

**rotateZ( Angle )**  
 Rotation about X axis by given angle. [Transformations Menu: Rotate]

**rotateLine( Line, Angle )**  
 Rotation about an arbitrary line.

**translateXYZ( XOffset, YOffset, ZOffset )**  
 Translation along X, Y, and Z by given amounts. [Transformations Menu: Translate XYZ]

**translateX( XOffset )**  
 Translation along X by given amount. [Transformations Menu: Translate]

**translateY( YOffset )**  
 Translation along Y by given amount. [Transformations Menu: Translate]

**translateZ( ZOffset )**  
 Translation along Z by given amount. [Transformations Menu: Translate]

**scaleUniform( Scale )**  
 Uniform scale along all three axes. [Transformations Menu: Uniform Scale]

**scaleXYZ( XScale, YScale, ZScale )**  
 Differential scale along all three axis. [Transformations Menu: XYZ Scale]

**scaleX( XScale )**  
 Scale along X axis. [Transformations Menu: Scale]

**scaleY( YScale )**  
 Scale along Y axis. [Transformations Menu: Scale]

**scaleZ( ZScale )**  
 Scale along Z axis. [Transformations Menu: Scale]

**scaleP3( P3Scale )**  
 Projective scale.

**genTran Mat16( a1, a2, ..., a16 )**  
 General transformation matrix (4 by 4).

**rotateXAxis( RotateToVector )**  
 Rotate current axis into given vector. [Transformations Menu: Axis to Vector]

**rotateYAxis( RotateToVector )**  
 Rotate current axis into given vector. [Transformations Menu: Axis to Vector]

**rotateZAxis( RotateToVector )**  
 Rotate current axis into given vector. [Transformations Menu: Axis to Vector]

**rotateVectorToXAxis( RotateToVector )**  
 Rotate given vector to become axis. [Transformations Menu: Vector to Axis]

**rotateVectorToYAxis( RotateToVector )**  
 Rotate given vector to become axis. [Transformations Menu: Vector to Axis]

**rotateVectorToZAxis( RotateToVector )**  
 Rotate given vector to become axis. [Transformations Menu: Vector to Axis]

**rotateXAxisWithY( XRotateVector, YRotateWithVector )**  
 Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateXAxisWithZ( XRotateVector, ZRotateWithVector )**



Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateYAxisWithX( YRotateVector, XRotateWithVector )**  
 Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateYAxisWithZ( YRotateVector, ZRotateWithVector )**  
 Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateZAxisWithX( ZRotateVector, XRotateWithVector )**  
 Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateZAxisWithY( ZRotateVector, YRotateWithVector )**  
 Rotate axis to given vector using second vector for extra degree of freedom. [Transformations Menu: Axis to Vector Orient]

**rotateVectorToXAxisWithY( XRotateVector, YRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rotateVectorToXAxisWithZ( XRotateVector, ZRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rotateVectorToYAxisWithX( YRotateVector, XRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rotateVectorToYAxisWithZ( YRotateVector, ZRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rotateVectorToZAxisWithX( ZRotateVector, XRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rotateVectorToZAxisWithY( ZRotateVector, YRotateWithVector )**  
 Rotate given vector to become axis using second vector for extra degree of freedom. [Transformations Menu: Vector to Axis Orient]

**rx( Angle )**  
 Same as rotateX.

**ry( Angle )**  
 Same as rotateY.

**rz( Angle )**  
 Same as rotateZ.

**tx( TransVal )**  
 Same as translateX.

**ty( TransVal )**  
 Same as translateY.

**tz( TransVal )**  
 Same as translateZ.

**sg( Factor )**  
 Same as scaleUniform.

**sxyz( ValX, ValY, ValZ )**  
     Same as scaleXYZ.  
**s3( ValX, ValY, ValZ )**  
     Same as scaleXYZ.  
**sx( Factor )**  
     Same as scaleX.  
**sy( Factor )**  
     Same as scaleY.  
**sz( Factor )**  
     Same as scaleZ.  
**extractMatrix( TransObj )**  
     Extracts a matrix from a transform, matrix descriptor, or instance.  
**appendDescriptor( TransObj, MatDescr )**  
     Add a new descriptor to the end of the list.  
**insertDescriptor( TransObj, N, MatDescr )**  
     Insert a new descriptor after the Nth one in the list.  
**deleteDescriptor( TransObj, N )**  
     Delete the Nth descriptor in the list.  
**replaceDescriptor( TransObj, N, MatDescr )**  
     Replace the Nth descriptor with the given new one.  
**indexDescriptor( TransObj, N )**  
     Returns the Nth descriptor in the list.

## A.17 Attribute Summary

**setAttr( Obj, AttrName, Value )**  
     Set the named attribute of the object to the specified value.  
**remAttr( Obj, AttrName )**  
     Remove the named attribute from the object.  
**getAttr( Obj, AttrName )**  
     Returns the value of the named attribute in the object.  
**stringAttr( InstanceName )**  
     Constructs a user-defined string attribute with the given name.  
**intAttr( InstanceName )**  
     Constructs a user-defined integer attribute with the given name.  
**floatAttr( InstanceName )**  
     Constructs a user-defined floating attribute with the given name.  
**attribute( Name, Value )**  
     Builds an attribute with given name and value.  
**attributeName( Attr )**  
     Extracts the name from an attribute.  
**attributeValue( Attr )**  
     Extracts the value from an attribute.  
**mkAdjacent( Srf1, Side1, Srf2, Side2 )**  
     Sets attributes on the two surfaces so that *Side1* of *Srf1* is adjacent to *Side2* of *Srf2*.

[Aggregates Menu: Make Adjacent]

**autoMkAdjacent**( *Srf1, Srf2, ..., SrfN* )

Attempt to set adjacencies automatically for a group of surfaces.

**autoMkAdjacentL**( *SrfList* )

Attempt to set adjacencies automatically for a group of surfaces.

## A.18 Miscellaneous Summary

**defParamType**( *TypeName, Param1, ...* )

Define a new parametric object type.

**saveModel**( *ObjStruct, File* )

Save the model and any intermediate constructions to the named file. [Save/Restore Menu: Save Model]

**saveModelInWindow**( *Window, File* )

Save all the geometry currently displayed in the given window. [Save/Restore Menu: Save Model In Window]

**restore**( *File* )

Restore the geometry from the named file. [Save/Restore Menu: Restore Model]

**showRestoredModel**()

Display the results of the last call to **restore**. [Save/Restore Menu: Show Restored Model]

**showNamedModel**()

Display just the named objects from the last call to **restore**.

**bindModelToSymtab**()

Put the named object from the last call to **restore** back into the

**dumpA1File**( *ObjStruct, File* )

Dump the objects into the named file for input to utility programs. [Object Utilities Menu: Dump to File]

**blinnParam**( *Specular, Diffuse, RefractIndex* )

Create an object containing the parameters for Blinn shading.

**distParam**( *Weight, Slope* )

Create an object containing the distribution parameters for shading.

**lightSource**( *Location, Intensity, Radius* )

Create an object containing the light parameters for ray tracing.

**subWorld**( *Objects* )

Attach the subworld attribute to a group of objects for the ray tracer.

**surfaceQuality**( *Color, RIndex, Specular, Diffuse, Transmit, Reflect, GExp, GDiff* )

Create an object containing the surface quality parameters for the ray tracer.

**reverseObj**( *Obj* )

Reverse the orientation of an object. [(Various Menus): Reverse]

**bboxObj**( *Obj* )

Returns a bounding box for any object which contains points.

**text**( *Location, String* )

Create a text string object. Can be displayed and dumped.

## A.19 Applications Summary

- angDimAttr**( *Point*, *Direction0*, *Direction1*, *AttrName*, *AttrVal*, ... )  
Create an angular dimension object.
- diaDimAttr**( *ArcOrCircle*, *AttrName*, *AttrVal* ) Create a diameter or radius dimension object.
- linDimAttr**( *Point0*, *Point1*, *AttrName*, *AttrVal* ) Create a linear dimension object.
- changeDimAttr**( *DimObj*, *AttrName*, *AttrValue* )  
Change a dimension attribute of a particular dimension object.
- changeDimDefaultAttr**( *DimType*, *AttrName*, *AttrValue* )  
Change the default of a dimension attribute for one dimension type.
- adoptDimDefaultAttrs**( *DimObj* )  
Copy the default attributes to a particular dimension object.
- counterBore**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth* )  
Create a counter-bore parametric type.
- inverseCounterBore**  
**inverseCounterBore**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth* ) Create an inverse counter-bore parametric type.
- counterSink**  
**counterSink**( *CtrPt*, *SinkDia*, *Angle*, *HoleDia*, *OvDepth* ) Create a counter-sink parametric type.
- counterDrill**  
**counterDrill**( *CtrPt*, *OutDia*, *InDia*, *OvDepth*, *InDepth*, *Angle* ) Create a counter-drill parametric type.
- slottedHole**  
**slottedHole**( *CtrPt*, *TotalLength*, *Width*, *Angle*, *Depth* ) Create a slotted hole parametric type.
- electricConnectorSet**  
**electricConnectorSet**( *CtrPt*, *LargeDia*, *Height*, *SmallDia*, *Depth*, *Space* ) Create an electric connector set parametric type.
- rectangularPocket**  
**rectangularPocket**( *UpperLeftCorner*, *TotalLength*, *Width*, *Depth*, *CornerRadius*, *BottomRadius* ) Create a rectangular pocket parametric type.
- linearPattern**  
**linearPattern**( *Object*, *StartPt*, *NumberOfObjects*, *Space*, *Angle* ) Create a linear pattern of manufacturing features.
- radialPattern**  
**radialPattern**( *Object*, *CtrPt*, *NumberOfObjects*, *StartAngle*, *EndAngle*, *PitchDia* )  
Create a radial pattern of manufacturing features.
- makeEndMill**( *KeyValue*, ... )  
Make an object representing an end mill with fields initialized as specified.
- makeDrill**( *KeyValue*, ... )  
Make an object representing a drill, with fields initialized as specified.
- MakeTap**( *KeyValue*, ... )  
Make an object representing a tapping tool, with fields initialized as specified.
- makeToolPosition**( *KeyValue* )

Make a tool position object as specified.

**makeNcStateVec( KeyValue, ... )**  
Make a state vector containing the specified information.

**ncCopyState( State )**  
Make a copy of the given state vector.

**ncNewTape( Name, Title )**  
Get ready to generate a new NC tape file.

**ncStateInit()**  
Initialize all the state variables for NC code generation.

**ncGen( TapeBlock )**  
Create a list of strings, the actual machine code generated for the block.

**ncFile( FileName )**  
Set the name of the NC output file.

**ncControl( GCodes )**  
Make an object containing the specified "G codes."

**ncSafeBlock( State )**  
Make a safe block object containing the given state vector.

**makeNcPosition( KeyValue, ... )**  
Move the NC tool to a certain position.

**ncSrfIso( Srf, SrfSpec, CrvSpec, NcState, Restriction )**  
Generate isoparametric curves as cutter path.

**ncSrfIsoZag( Srf, SrfSpec, CrvSpec, NcState, Restriction )**  
Generate isoparametric curves as cutter path, alternating cutter direction.

**ncChordParms( Crv, Spacing )**  
Compute a list of parameter values for a curve with given point spacing.

**ncCrvMill( Crv, Spec, Contact, Depth, DoCut, NcState )**  
Move the cutter along a single curve in space.

**ncGenProfile( Profile, ClimbMill, Depth, RoughOffset, NcState )**  
Generate NC path for a profile curve made up of straight lines and arcs.

**ncProfileOffset( Profile, ClimbMill, Offset )**  
Generate a new profile offset from the original one.



## Appendix B. Summary of Menu Entries

This appendix describes briefly all of the menu entries available from the graphical user interface. The menu-driven interface is just a convenient wrapping for the command-driven interface, so most menu selections correspond to a command which is described in full detail in the main body of the manual. For this reason, only a short description of each item is given here, and relevant global variables, symbols, keywords, or functions described elsewhere are listed in parentheses after the description.

The graphical user interface has a two-level menu structure. All menu actions are grouped into one of the menus which is available from the main menu.

### Main Menu

#### Object Utilities

Utilities for manipulating objects.

#### Screen Utilities

Utilities for controlling the display.

#### Pick Object Type

Setting the type for the next object to be picked.

#### Save/Restore

Saving the data to a file, or restoring from a file.

**Numbers** Operations that generate numbers.

**Points** Operations that generate points.

**Vectors** Operations that generate vectors.

#### Lines & Planes

Operations that generate lines and planes.

#### Arcs & Circles

Operations that generate circular arcs and full circles.

#### Basic Curves

Basic operations for making curves.

#### Basic Surfaces

Basic operations for making surfaces.

**Primitives** Operations for constructing solid primitives.

#### Curve Fitting

Generating curves that interpolate data.

#### Shape Operations

Operations for changing the shape of a surface.

#### Aggregates

Grouping and instancing objects.

#### Transformations

Ways to specify scales, translations, and rotations.

**Keywords** Some common keywords used by other operations.

#### Fitting Keywords

Keywords used by the curve fitting operations.

#### Rbox Keywords

Keywords used by the rounded box construction.

**Parminfo** Ways to define the parametric information for a surface.

**Spline Functions**

Curves with special properties for interpreting as functions.

The "Object Utilities" menu contains functions for editing objects and controlling the display of objects.

**Object Utilities Menu**

**Modify Object**

Change the parameters which were used to construct an object.

**Modify Point**

Change the location of a point (separate from Modify Object because points are hard to pick).

**Replace Object**

Replace an object with a different one.

**Replace Point**

Replace a point with a different one (separate from Replace Object because points are hard to pick).

**Show Object**

Display an object given its name. (**show**)

**Show Prereqs**

Show the objects which were used to construct another object. (**showPrereqs**)

**Forget Object**

Throw away an object. (**forgetName**)

**Unshow Object**

Remove an object from the display. (**unshow**)

**Unshow Prereqs**

Remove the objects which were used to construct an object from the display. (**unshow-Prereqs**)

**Unshow List**

Remove several objects from the display. (**unshow**)

**Reshow Object**

Redraw an object. (**reshow**)

**Reshow List**

Redraw several objects. (**reshow**)

**Highlight Object**

Highlight an object on the display. (**highlight**)

**UnHighlight Object**

Un-highlight an object on the display. (**unHighlight**)

**Name Object**

Give an object a name. ("**:=**" or "**^=**")

**Dump to File**

Write the data to file in Alpha\_1 text file format. (**dumpA1File**)

**Print Object**

Print a text representation of the object.

**Describe Object**



Print **everything** about the object. (**describe**)

The "Screen Utilities" menu contains functions for manipulating windows on the display and for controlling the appearance of the graphics.

### Screen Utilities Menu

#### Curve Polys

Turn display of control polygons for curves on or off. (**setCrvPolys**)

#### Smooth Curves

Turn display of smooth curves on or off. (**setSmoothCurves**)

#### Curve Fineness

Set the resolution at which curves will be displayed. (**setCrvFineness**)

#### Surface Meshes

Turn display of control meshes for surfaces on or off. (**setSrfMeshes**)

#### Isoparametric

Turn display of isoparametric lines for surfaces on or off. (**setIsoLines**)

#### Surface Fineness

Set the resolution at which isoparametric lines for surfaces will be spaced. (**setSrfFineness**)

#### Surface Normals

Turn display of surface normals on or off. (**setSrfNormals**)

#### Reverse Normals

Turn reversing of surface normal display on or off. (**setReverseNormals**)

#### New Window

Create a new window on the display. (**newWindow**)

#### Select Window

Select a particular window in which geometry will be displayed. (**selectWindow**)

#### Deselect Window

Disable display of new geometry in a particular window. (**deselectWindow**)

#### Clear Window

Clear the contents of a particular window. (**clearWindow**)

#### Remove Window

Remove a particular window from the display. (**removeWindow**)

The "Pick Object Type" menu allows you to specify what kind of object you are about to try to pick. Normally, this is done automatically, so that if only one kind of object is allowed as an argument to a function, only that kind of object is *pickable*. However, many operations have arguments where the value may be one of several object types. If the objects happen to look the same on the display, these functions allow you to specify which one you want. There are cases, however, where the underlying representation for two objects is the same type. Arcs, for example, are sent to the display as curves, and are indistinguishable from curves as far as the display is concerned. In these cases, the "Pick Object Type" menu will not help, and you will just have to figure out how to unshow some of the objects from the display in order to do the picking.

### Pick Object Type Menu

- Point      Restrict the next pick to point objects.
- Vector     Restrict the next pick to vector objects.
- Line       Restrict the next pick to line objects.

<b>Plane</b>	Restrict the next pick to plane objects.
<b>Polyline</b>	Restrict the next pick to polyline objects.
<b>Polygon</b>	Restrict the next pick to polygon objects.
<b>Arc</b>	Restrict the next pick to arc objects.
<b>Circle</b>	Restrict the next pick to circle objects.
<b>Curve</b>	Restrict the next pick to curve objects.
<b>Surface</b>	Restrict the next pick to surface objects.
<b>String</b>	Restrict the next pick to string objects.

The "Save/Restore" menu contains a few functions which allow the geometry constructed in one `shape_edit` session to be saved and reloaded for further editing in another session.

#### Save/Restore Menu

##### Save Model

Save a model to the named file for later continuation of work. (`saveModel`)

##### Save Model in Window

Save everything in a given window for later continuation of work. (`saveModelInWindow`)

##### Restore Model

Load from a file a model which was previously saved. (`restore`)

##### Show Restored Model

Display the last model which was restored. (`showRestoredModel`)

The "Numbers" menu provides some easy ways to derive numbers from geometry which appears on the display, or to select commonly used numbers. The numbers 0, 1, 30, 45, 60, and 90 appear on the menu. There are 10 available slots for users to add their own numeric constants if desired.

#### Numbers Menu

##### X Coordinate

Extract the X coordinate of a point or vector. (`ptX` or `vecX`)

##### Y Coordinate

Extract the Y coordinate of a point or vector. (`ptY` or `vecY`)

##### Z Coordinate

Extract the Z coordinate of a point or vector. (`ptZ` or `vecZ`)

##### W Coordinate

Extract the W coordinate of a point. (`ptW`)

##### Nth Coordinate

Extract the coordinate of a point or vector given by an index.

##### Vector Length

Compute the length of a vector. (`vecLength`)

##### 2 Pt Distance

Compute the distance between two points. (`distPtPt`)

##### Pt Line Distance

Compute the perpendicular distance from a point to a line. (`distPtLine`)

##### Pt Plane Distance

Compute a signed distance from a point to a line. (`signedDistPtPlane`)

##### Angle Between

- Compute the angle between two lines or two vectors.
- Radius** Extract the radius from an arc or circle. (**radiusOfArc** or **circleRad**)
- Negate** Negate the value of a number. (Unary "-")
- Plus** Add two numbers. ("+")
- Times times2**  
Multiply two numbers. ("\*")
- Divide quotient**  
Divide a number by another. ("/")
- Add Constant addConstant**  
Add a new constant to the numbers menu.
- The "Points" menu provides a number of ways to make points.
- Points Menu**
- Coordinates**  
Construct a point given the coordinates. (**pt**)
- Projective By Coords**  
Construct a projective point given the coordinates. (**projPt**)
- Origin** The predefined point at the origin. (**Origin**)
- Offset** Construct a point offset from another by a given vector. (**ptOffset**)
- Scaled Offset**  
Construct a point offset from another by a scaled vector. (**ptScaledOffset**)
- 2 Pt Interp**  
Construct a new point on the line joining two other points. (**ptInterp**)
- 2 Lines Intersect**  
Construct a point at the intersection of two lines. (**ptIntersect2Lines**)
- Pt Project to Line**  
Construct a new point by projecting it onto a line. (**projectPtOntoLine**)
- Reflect Point**  
Reflect a point through a plane or about an axis. (**reflect**)
- 3 Planes** Construct the point at the intersection of three planes. (**ptFrom3Planes**)
- Pt Project to Plane**  
Construct a point by projecting it onto a plane. (**ptProjPtDirPlane**)
- Center Of** Extract the center point of a circle or circular arc. (**centerOfArc** or **circleCtr**)
- Arc Start** Extract the first point of an arc. (**arcStart**)
- Arc End** Extract the last point of an arc. (**arcEnd**)
- 2 Circles Intersect**  
Construct one of the points at the intersection of two circles. (**ptIntersect2Circles**)
- Curve Eval**  
Construct a point by evaluating a curve. (**crvEval**)
- Bbox Min** Construct a point which is the minimum of a bounding box for an object.
- Bbox Max calcBboxMax**  
Construct a point which is the maximum of a bounding box for an object.
- PolyLine** Build a sequence of points into a polyline. (**polyLine**)
- Polygon** Build a sequence of points into a polygon. (**polygon**)

**Point Mesh**

Build a sequence of points into a control mesh. (**ctlMesh**)

The "Vectors" menu provides a variety of ways to create vectors.

**Vectors Menu****Coordinates**

Construct a vector from the given coordinates. (**vec**)

**X Direction**

The predefined vector in the direction of the X axis. (**Xdir**)

**Y Direction**

The predefined vector in the direction of the Y axis. (**Ydir**)

**Z Direction**

The predefined vector in the direction of the Z axis. (**Zdir**)

**Unit Vector**

Construct a normalized vector. (**unitVec**)

**Scale**

Scale a vector. (**vecScale**)

**2 Points**

Construct a vector in the direction from one point to another. (**vecFrom2Pts**)

**At Angle**

Construct a vector at a given angle to the horizontal. (**vecAtAngle**)

**Add Vectors**

Vector addition. (**vecPlus**)

**Subtract Vectors**

Vector subtraction. (**vecMinus**)

**2 Vec Interp**

Weighted sum of two vectors. (**vecInterp**)

**Line Direction**

Construct a vector in the direction of a given line. (**dirOfLine**)

**Perp Line Direction**

Construct a vector perpendicular to the direction of a given line. (**dirPerpLine**)

**2 Vector Perp**

Compute the cross product of two vectors. (**crossProd**)

**Plane Normal**

Extract the normal vector of a given plane. (**planeNormal**)

**Project Vec to Plane**

Construct a vector by projecting one onto a plane. (**vecProjVecOntoPlane**)

**Curve Derivative**

Construct a vector by evaluating the derivative of a curve. (**crvDerivEval**)

**Curve N Derivative**

Construct a vector by evaluating the Nth derivative of a curve. (**crvDerivEval**)

The "Lines" menu contains a number of functions for creating lines.

**Lines Menu****X Axis**

The predefined line corresponding to the X axis. (**Xaxis**)

**Y Axis**

The predefined line corresponding to the Y axis. (**Yaxis**)

**Vertical**

Construct a vertical line. (**lineVertical**)

- Horizontal** Construct a horizontal line. (**lineHorizontal**)
- 2 Points** Construct a line through two points. (**lineThru2Pts**)
- Point & Angle**  
Construct a line through a point at a given angle to the horizontal. (**linePtAngle**)
- Point & Direction**  
Construct a line through a point in a given direction. (**linePtVec**)
- Point Parallel**  
Construct a line through a point parallel to another line. (**linePtParallel**)
- Offset Line**  
Construct a line at a given offset distance from another line. (**lineOffsetFromLine**)
- Point & Circle**  
Construct a line through a point and tangent to a circle. (**linePtCircle**)
- 2 Circles** Construct a line tangent to two circles. (**lineTan2Circles**)
- XY Plane** The predefined XY plane. (**XYPlane**)
- YZ Plane** The predefined YZ plane. (**YZPlane**)
- XZ Plane** The predefined XZ plane. (**XZPlane**)
- Point & Normal**  
Construct a plane through a point with a given normal. (**planeThruPtWithNormal**)
- Offset Plane**  
Construct a plane at a given offset distance from another plane. (**planeOffsetByDelta**)
- Reverse** Reverse the orientation of an object. (**reverseObj**)
- The "Arcs and Circles" menu contains functions for creating both circular arcs and complete circles.
- Arcs and Circles Menu**
- Radius & 2 Lines**  
Construct an arc with a given radius and tangent to two lines. (**arcRadTan2Lines**)
- 3 Lines** Construct an arc tangent to three lines. (**arcTan3Lines**)
- 3 Points** Construct an arc through three point. (**arcThru3Pts**)
- End Center End**  
Construct an arc given the two end points and the center. (**arcEndCenterEnd**)
- Cut From Circle**  
Construct an arc by cutting it from a circle with a line. (**arcCutFromCircle**)
- Radius Circle Line**  
Construct an arc with a given radius tangent to a circle and a line. (**arcRadTanToCircleAndLine**)
- Radius 2 Circles**  
Construct an arc with a given radius tangent to two circles. (**arcRadTan2Circles**)
- Reverse** Reverse the orientation of an object. (**reverseObj**)
- Other Arc** Construct an arc which is the completion of the circle for a given arc. (**OtherArc**)
- Unit Circle**  
The predefined curve which represents the unit circle. (**UnitCircle**)
- Center Radius**  
Construct a circle given the center and radius. (**circleCtrRad**)

**Center Point**

Construct a circle given the center and a point on the circle. (**circleCtrPt**)

**Radius 2 Circles**

Construct a circle with a given radius tangent to two circles. (**circleRadTan2Circles**)

The "Basic Curves" menu contains some simple functions for creating curves.

**Basic Curves Menu**

**Curve** Construct a curve by specifying the control points and parametric information. (**curve**)

**From Arc** Convert an arc into a curve. (**crvFromArc**)

**From Circle**

Convert a circle into a curve. (**crvFromCircle**)

**Profile** Construct a curve by chaining together a sequence of points, arcs, and other curves. (**profileList**)

**Reflect** Reflect a curve through a plane or about an axis. (**reflect**)

**Offset** Compute an offset to a given curve. (**qOffset**)

**Reverse** Reverse the orientation of an object. (**reverseObj**)

**Open End Condition**

Convert a curve to have open end conditions. (**curveOpen**)

**Refine** Compute a refined curve by specifying new elements of the knot vector. (**cRefine**)

**Raise Degree**

Raise the polynomial degree of a curve. (**raiseDegree**)

**Curve Region**

Extract a region of a curve. (**regionFromCrv**)

**Project Crv To Plane**

Project a curve onto a plane. (**crvFromCrvProjOntoPlane**)

**From Surface Mesh**

Extract a row or column curve from a surface control mesh. (**crvFromSrf**)

**In Surface** Extract a curve which lies in a surface. (**crvInSrf**)

**Surface Boundary**

Extract one of the boundary curves of a surface. (**getBoundary**)

**Tangent Lines**

Construct a curve by specifying places where it is tangent. (**crvTangLines**)

The "Basic Surfaces" menu contains some simple functions for creating surfaces.

**Basic Surfaces Menu**

**Surface** Construct a surface by specifying its control mesh and parametric information. (**surface**)

**Flat** Construct a flat surface. (**flatSrf**)

**Bounded Flat**

Construct a flat surface of a given size. (**flatSrfBounds**)

**2 Pt Extrude**

Construct a surface by extruding a curve, specifying two base points. (**extrude**)

**Direction Extrude**

- Construct a surface by extruding a curve in a given direction. (**extrudeDir**)
  - Ruled** Construct a surface which is the ruled surface between two curves. (**ruledSrf**)
  - Of Revolution** Construct a surface of revolution from a curve and a line. (**srfOfRevolution**)
  - Profile & Section** Construct a surface from a profile curve, a center axis, and a cross section. (**srfAxis-ProfileSection**)
  - 4 Boundary Curves** Construct a surface by blending four boundary curves. (**boolSum**)
  - Tube** Construct a surface by sweeping a circle along a path. (**circTubeConstantWidth**)
  - Sweep** Construct a surface by sweeping a cross section along a path. (**sweepConstantWidth**)
  - Profiled Sweep** Construct a surface by sweeping a cross section along a path using a profile curve to control the scaling of the cross section. (**sweepWithProfile**)
  - Refine** Refine a surface in one direction by specifying a set of additional knot values. (**sRefine**)
  - Open End Condition** Convert a surface to open end conditions. (**surfaceOpen**)
  - Reverse** Reverse the orientation of an object. (**reverseObj**)
  - Surface Region** Extract a region from a surface in one direction. (**regionFromSrf**)
- The "Primitives" menu contains functions for creating solid primitives.
- Primitives Menu**
- Box With Vecs** Construct a box from three direction vectors and a base point. (**box**)
  - Box With Planes** Construct a box from six planes. (**rboxFrom6Planes**)
  - Wedge** Construct a wedge from direction vectors and a base point. (**rightAngleWedge**)
  - Cylinder** Construct a right circular cylinder. (**rightCirCylinder**)
  - Rounded Cylinder** Construct a right circular cylinder with rounded top and bottom edges. (**roundRight-CirCylinder**)
  - Truncated Cone** Construct a truncated right circular cone. (**truncRightCone**)
  - Rounded Truncated Cone** Construct a truncated right circular cone which has rounded top and bottom edges. (**roundTruncCirCone**)
  - Sphere** Construct a sphere. (**sphere**)
  - Unit Sphere** The predefined sphere at the origin with radius 1.0. (**UnitSphere**)
  - Torus** Construct a torus. (**torus**)
  - Rounded Box** Construct a box with rounded edges. (**rboxFrom6Planes**)

**Set Rbox Radius**

Set the radius on an edge of a rounded box. (**setRboxRadius**)

**Set Rbox Faces Open**

Set one face of a rounded box to be open. (**SetRboxFacesOpen**)

**Offset Rbox**

Construct a rounded box as an offset of an existing one. (**rboxOffsetFromRbox**)

The "Curve Fitting" menu contains a few functions for fitting curves to sets of point data.

**Curve Fitting Menu****Complete Cubic**

Perform a complete cubic interpolation on a set of points. (**completeCubicInterp**)

**Cubic**

Perform an interpolation to generate a cubic curve given parameter values and a set of points. (**cubicCrvFromParametersAndPositions**)

**Using Parminfo**

Perform an interpolation to generate a curve from parametric information and a set of points. (**crvFromParmInfoAndPositions**)

**Periodic Complete Cubic**

Perform a complete cubic interpolation on a set of points to produce a periodic curve. (**periodicCompleteCubicInterp**)

The "Shape Operations" menu contains some more advanced operations which allow surfaces to be modified and shaped in various ways.

**Shape Operations Menu****Offset**

Compute an approximate offset to a surface. (**vOffset**)

**Lift**

Generate a lifting of a surface. (**lift**)

**Thicken**

Construct a linear join between two surfaces, which goes around the entire boundary. (**linearSrfEdge**)

**Join Linear**

Construct a linear join between two surface edges. (**singleLinearSrfEdge**)

**Join Quadratic**

Construct a quadratic join between two surface edges. (**singleQuadraticSrfEdge**)

**Join Cubic**

Construct a cubic join between two surface edges. (**singleCubicSrfEdge**)

**Add Flex**

Add more control points to a region of a surface. (**addFlex**)

**Isolate Region**

Isolate a region of a surface from changes which occur outside. (**isolateRegion**)

**Feature Line**

Add a potential discontinuity to a surface. (**featureLine**)

**Bend**

Bend a surface. (**bend**)

**Stretch**

Stretch a surface. (**stretch**)

**Taper**

Taper a surface. This is more restricted than the one in the command driven interface; the tapering function must be given as a spline curve which is best created from the "Spline Functions" menu. (**taperWithSplineFn**)

**Twist**

Twist a surface. This is more restricted than the one in the command driven interface; the twisting function must be given as a spline curve which is best created from the



“Spline Functions” menu. (**twistWithSplineFn**)

**Flatten** Flatten a surface against a plane. (**flattenSrf**)

**Warp** Warp (add a bump to) a surface. (**warp**)

**Skeletal Warp**  
Warp a surface along a line. (**skelWarp**)

**Region Warp**  
Warp a surface inside a given region. (**RegionWarp**)

The “Aggregates” menu provides ways to group objects into higher level entities, and ways to make and manipulate instances of objects.

### Aggregates Menu

**Instance** Create an instance of an object in a new location or orientation. (**instance**)

**Replace Instance**  
Replace the object in an instance with a new one, keeping the same transformation. (**replaceObject**)

**Group** Create a group of objects. (**group**)

**Add To Group**  
Add a new object to an existing group. (**addToGroup**)

**Delete From Group**  
Delete an object from a group. (**deleteFromGroup**)

**Shell** Form a shell object from a set of surfaces. (**shell**)

**Make Adjacent**  
Declare adjacencies between edges of surfaces. (**mkAdjacent**)

The “Transformations” menu allows standard graphics transformations to be specified in a variety of ways.

### Transformations Menu

**Translate** Translate along one coordinate axis. (**translateX, translateY, translateZ**)

**Pt Translate**  
Translate origin to a given point. (**translateXYZ**)

**Translate XYZ**  
Translate along a given vector. (**translateXYZ**)

**Scale** Scale relative to a coordinate axis. (**scaleX, scaleY, scaleZ**)

**Uniform Scale**  
Scale uniformly along all axes. (**scaleUniform**)

**XYZ Scale**  
Scale all three axes independently. (**scaleXYZ**)

**Rotate** Rotate about an axis. (**rotateX, rotateY, rotateZ**)

**Axis to Vector**  
Rotate an axis into a given vector. (**rotateXAxis, etc.**)

**Vector To Axis**  
Rotate a given vector into an axis. (**rotateVectorToXAxis, etc.**)

**Axis to Vector Orient**  
Rotate an axis into a given vector, with a second axis also specified. (**rotateXAxisWithY, etc.**)

**Vector To Axis Orient**

Rotate a given vector into an axis, with a second vector also specified. (**rotateVector-ToXAxisWithY**, etc.)

**Make Transform**

Group a set of matrix descriptors into a transform object. (**transformL**)

**Transform Object**

Map an object through a transformation matrix. (**ObjTransform**)

The "Keywords" menu provides easy access to commonly used keywords.

**Keywords Menu**

<b>Yes</b>	A boolean true value. ( <b>T</b> )
<b>No</b>	A boolean false value. ( <b>Nil</b> )
<b>Row</b>	Specification for row direction of a surface. ( <b>ROW</b> )
<b>Column</b>	Specification for column direction of a surface. ( <b>COL</b> )
<b>X</b>	Specification of X coordinate axis. ( <b>'X</b> )
<b>Y</b>	Specification of Y coordinate axis. ( <b>'Y</b> )
<b>Z</b>	Specification of Z coordinate axis. ( <b>'Z</b> )
<b>Top</b>	Specification for top edge of a surface. ( <b>'TOP</b> )
<b>Bottom</b>	Specification for bottom edge of a surface. ( <b>'BOTTOM</b> )
<b>Left</b>	Specification for left edge of a surface. ( <b>'LEFT</b> )
<b>Right</b>	Specification for right edge of a surface. ( <b>'RIGHT</b> )

The "Fitting Keywords" menu contains keywords which are used only in the curve fitting functions.

**Fitting Keywords Menu**

<b>Uniform</b>	Specification of uniform knot vector for curve fitting. ( <b>Uniform</b> )
<b>ChordLength</b>	Specification of chord length knot vector for curve fitting. ( <b>ChordLength</b> )
<b>ParamByX</b>	Specification of knot vector parametrized by X coordinates for curve fitting. ( <b>Param-ByX</b> )
<b>ParamByY</b>	Specification of knot vector parametrized by Y coordinates for curve fitting. ( <b>Param-ByY</b> )
<b>ParamByZ</b>	Specification of knot vector parametrized by Z coordinates for curve fitting. ( <b>Param-ByZ</b> )
<b>Radial</b>	Specification of knot vector parametrized radially for curve fitting. ( <b>Radial</b> )

The "Rbox Keywords" menu contains keywords which are only used when creating rounded edge boxes.

**Rbox Keywords Menu**

<b>Right Face</b>	Specification of the right face of a box. ( <b>'RFace</b> )
<b>Left Face</b>	Specification of the left face of a box. ( <b>'LFace</b> )

**Up Face** Specification of the up (top) face of a box. ('UFace)

**Down Face** Specification of the down (bottom) face of a box. ('DFace)

**Front Face** Specification of the front face of a box. ('FFace)

**Back Face** Specification of the back face of a box. ('BFace)

**Edge Between** Specification of the edge between two faces. ('RDedge, etc.)

**All Edges** Specification of all the edges of a box. ('AllEdges)

The "Parminfo" Menu contains the symbols and functions used for creating parametric information for spline curves and surfaces. It also has some predefined sets for the most commonly used cases.

#### Parminfo Menu

**From Curve** Extract the parametric information from an existing curve. (cParmInfo)

**From Surface U (ROW) dir** Extract the U parametric information from an existing surface. (rowParmInfo)

**From Surface V (COL) dir colParmInfo** Extract the V parametric information from an existing surface. (colParmInfo)

**Open, uniform, quadratic** Create a parmInfo object with quadratic order, open end conditions, and a uniform knot vector.

**Open, uniform, cubic** Create a parmInfo object with cubic order, open end conditions, and a uniform knot vector.

**Open, chord, quadratic** Create a parmInfo object with quadratic order, open end conditions, and a chord length knot vector.

**Open, chord, cubic** Create a parmInfo object with cubic order, open end conditions, and a chord length knot vector.

**Create Parminfo** Create the parametric information for a curve or surface. (parminfo)

**Linear** Specify linear order for a spline. (LINEAR)

**Quadratic** Specify quadratic order for a spline. (QUADRATIC)

**Cubic** Specify cubic order for a spline. (CUBIC)

**Quartic** Specify quartic order for a spline. (QUARTIC)

**Quintic** Specify quintic order for a spline. (QUINTIC)

**Open** Specify open end conditions for a spline. (EC\_OPEN)

**Floating** Specify floating end conditions for a spline. (EC\_FLOATING)

**Periodic** Specify periodic end conditions for a spline. (EC\_PERIODIC)

**Uniform** Specify uniform knot vector for a spline. (KV\_UNIFORM)

**Bezier** Specify Bezier knot vector for a spline. (KV\_BEZIER)

**Chord** Specify chord length knot vector for a spline. (KV\_CHORD)

**Number List**

Specify a knot vector explicitly. (**knot\_vector**)

The "Spline Functions" menu contains operations which create spline function curves for use by the taper and twist shape operations.

**Spline Functions Menu**

**Constant** Create a spline function with constant value. (**constantSplineFn**)

**Linear** Create a spline function which changes linearly. (**linearSplineFn**)

**Approximate Values**

Create a spline function which approximates a given set of points. (**approxValuesSplineFn**)

**Interpolate Values**

Create a spline function which interpolates a given set of points. (**interpValuesSplineFn**)



## Appendix C. Device Specific Window Managers

This appendix includes information about details of window management and other `shape_edit` display facilities which are device specific.

### C.1 Silicon Graphics Iris Display

In order to use the iris as a display device, one should first log into the iris and start the window manager. To start the window manager, type `"mex -d"`. The initial configuration of the window manager is dictated by the `".mexrc"` file. Copy the `".mexrc"` file provided with Alpha\_1 into your home directory. (For additional information, see the "Window Manager" section of the "Iris Users Guide".)

The Iris device is accessed in `shape_edit` using the device name "iris", as in

```
grab iris;
```

or

```
drop iris;
```

#### Creating Windows

A little status message window is created when the iris device is first grabbed from `shape_edit` or `view`. A green wire-frame rectangle is displayed, showing the size of the message window. Position the cross shaped cursor where you want the lower left corner of the message window to be placed (the window outline follows along), then press the left mouse button.

When a `newWindow` command is given, a right angle cursor will appear on the screen. Press the left button down where you want the upper left of the window to appear. Drag the mouse to the desired position of the lower right corner of the window and let up on the button. The window will be drawn in a red border to indicate that it is the current window where mouse and keyboard events will be sent.

#### Window Management

The `".mexrc"` file provided with Alpha\_1 gives a specific meaning to some mouse events and keyboard buttons.

The "no scroll" key is a Mex "shift" key. Hold it down while using the mouse to manage windows. When the "no scroll" button is pressed, the cursor changes to a little arrow with three overlapping rectangles symbolizing windows.

To pop a window on top of any windows that may be covering it and send further events to it, move the cursor onto the window, hold down the "no scroll" key, and click the left mouse button. The red border will move to that window.

To resize a window (move just one of its corners), put the cursor on the window, within about the size of the cursor from the corner. Hold down the "no scroll" key and press the middle mouse button. The outline of the window will be drawn in green. Hold the middle mouse button down and drag the corner of the window to a new position, then release the middle mouse button.

To move a window, put the cursor on the window, more than the size of the cursor away from a corner of the window. Hold down the "no scroll" key and press the middle mouse button, and the outline of the window will be drawn in green. The difference between move and resize is that the whole window outline will follow the cursor, rather than just one corner. Drag the window to the new location, then release the middle mouse button.



To attach events to an unattached window without popping it, just click the right mouse button in the window. The "no scroll" key is not needed in this case. (This actually selects the default function from the Mex window management menu.)

### Mex Menus

The right mouse key is always used for menu item selection under Mex. Different menus are presented, depending on what window the cursor points to when the right mouse button is first pressed. Hold down the right mouse button while you move the cursor to the desired menu item, then release the button. The menu item under the cursor is highlighted. If you don't want to select anything, just move the cursor off the menu and release the button.

When the cursor is not in any window, the Mex main menu is presented. It allows you to create a new shell window or exit Mex. You should exit Mex when you log out, because anybody could still create a shell window which would be logged in as you if Mex were left running.

In an unattached window (one without the red border), or an attached window when the "no scroll" key is pressed, the Mex window management menu allows you to attach, pop, push, resize, move, or kill a window. Killing a window actually means killing the process that created the window, so if you kill one `shape_edit` window they all go away. (But the `shape_edit` is a separate process, so it stays around.)

### Shape\_edit Menus

If you are running the `shape_edit` Graphical User Interface, and are attached to a `shape_edit` window, the `shape_edit` menus are presented. The default menu structure has a main menu full of submenu items (a little arrow is shown on the right side of each submenu item.)

To select an item from a submenu, hold down the right mouse button and move the cursor down the main menu to the desired submenu item. When you move the cursor off either the left or right side of a submenu item, the submenu pops up.

In the GUI, many operations know the type of object they expect as an operand, and set the appropriate submenu to pop up when the right mouse button is pressed. The message window title changes to "Current Menu" when there is a menu set loaded, and the default menu name is shown in the message window. The **Keypad Comma Key** (on the numeric and PF key pad to the right of the regular keyboard keys) toggles between the main menu and the last submenu. You can get from a submenu to the main menu by hitting Keypad Comma before pressing the right mouse button. Similarly, Keypad Comma is a quick way to get back to the last submenu you used from the main menu.

### Viewing

3D viewing is controlled by the mouse, with keyboard keys setting the viewing mode. The viewing mode is echoed by an appropriate cursor shape. Viewing parameters are changed by holding the left mouse button down and dragging the mouse. Below is a summary of all the keys that are reserved for viewing and their various functions.

**Up Arrow** X and Y rotate. Horizontal changes in the mouse position rotate the image left and right. Vertical changes in the mouse position rotate the image up and down.

**Down Arrow** X and Y translate. The image is dragged to follow the cursor movement.

**Left Arrow** Uniform scale. Moving the mouse to the right increases the image size. Moving the mouse to the left decreases the image size.



**Right Arrow**

Z rotate. Position the cursor at least an inch from the center of the window. Moving the cursor relative to the center of the window rotates the image.

**PF1** Z translate. Moving the mouse to the right causes the image to be translated back (away from you) in Z. Moving to the left causes the image to be translated forward (toward you) in Z.

**PF2** Translate near clipping plane. Moving the mouse to the right pushes the front clipping plane away from the eye. Moving to the left pulls the front clipping plane toward the eye.

**PF3** Translate far clipping plane. Moving the mouse to the right pushes the far clipping plane away from the eye. Moving to the left pulls the far clipping plane toward the eye.

**Setup** Reset view. Goes back to an identity matrix.

**Ctl-Setup** Save matrix. A copy of the current viewing matrix is saved for later use.

**Shift-Setup** Restore matrix. The saved viewing matrix is put in the current window. Note that the viewing matrix could have been saved from the same or a different window.

**Break** Send matrix. Sends a copy of the current viewing matrix back to `shape_edit` or `view` as an input event.

**Backspace** Leave viewing mode. The default arrow cursor comes back, which is useful as a pointer.

**Tab** Toggle depth cueing. The display will update more quickly if depth cueing is turned off.

**Escape** Toggle orthographic and perspective projections.

**Picking and locating**

The middle mouse button is used for picking when unshifted, and locating when one of the shift keys is held down.

When picking an object on the screen, press the middle mouse button down. The cursor changes to a small square the size of the pick region. Move the square cursor over part of the object which is drawn on the screen, and release the middle mouse button. The identity of the object is transmitted to `shape_edit` as a pick event.

Locators are actually lines in the object space which pass through the eye point and a point on the screen. They are currently interpreted as requests to create a point in the GUI, so the ray is intersected with a construction plane. To locate a point, hold down one of the shift keys and press the middle mouse button down. The cursor changes to a small marker like a plus sign, which is the shape used for displaying points. Move the point marker to the desired location, and release the middle mouse button. A locator event is transmitted to `shape_edit`.

With practice, you will find you often do not need to see the cursor shape. Then you can do picks and locates as a single click of the middle mouse button, rather than as a two-step operation.

**Valuator**

Valuator mode combines the keypad digit keys and the mouse to type numbers in to the GUI and/or to adjust them smoothly. The cursor is changed to a "V" while in valuator mode, the title of the message window is changed to "Valuator", and the current value of the valuator is shown in the message window.

The keypad digits, decimal point, and "minus" keys are used for typing numbers. When you enter valuator mode by typing one of them, the current value starts out as "0.000". (One bug is that the minus key currently just negates the current value, which is pretty ineffective if it is zero. Don't start with the minus key until that is fixed.)

You can also re-enter valuator mode with the previous value by typing the keypad "Enter" key.

Once in valuator mode, typing a digit key changes one digit of the number, clears the digits to the right to zeros, and advances to the next digit. Typing the "del" key deletes a digit from the number. The "current digit" to be changed is shown by a pair of vertical arrows in the message window, and the cursor X position.

The mouse is used as a 2D slider with the X (right to left) position of the cursor controlling the scale and Y (top to bottom) motion with the left mouse button pressed incrementally changing the value. In effect, the screen is divided into vertical slider strips, one for each digit. The digit changes rapidly at the left edge of the strip, and slowly at the right.

A number event is sent to the GUI by clicking the middle mouse button, or by another "Enter" key press which also leaves valuator mode. ASCII (white) keyboard keys, "backspace", viewing key presses, menu selection, etc. leave valuator mode without transmitting a number.

A little pop-up menu controls the valuator precision. While in valuator mode, move the cursor to the Valuator (message) window and press the right mouse button to get the valuator menu. You can increase or decrease the number of digits to the left or right of the decimal point. (The default is two digits before the decimal point and three after.)

Note that the Iris (not the newer 4D) line is limited to single precision, so many numbers with more than 5 or 6 digits cannot be represented. You can see this if you try to drag the least significant digits of long numbers with the mouse.

### Strings

There is no special provision for typing strings such as object names or filenames in the Iris device handler at this time. You can attach to the shell window from which **shape\_edit** was run by clicking the right mouse button on it. Type a string enclosed in double quotes and followed by a semicolon, for instance:

```
"temp.r";
```

Finally, hit the "Return" key. Usually, you will then need to attach back to a **shape\_edit** window by another click of the right mouse button.

## C.2 X Window System Displays

The X window system is accessed from **shape\_edit** using the device name "xgen" as in

```
grab xgen;
```

or

```
drop xgen;
```

When xgen is grabbed, an X window is created (subsequently called the "viewing window") which is a window of subwindows which emulate slider bars and buttons. (See sections below on how to create this window or specify a default.) These slider bars and buttons are used to control the viewing transformations, provide information for the GUI and otherwise control the display. These functions are described in detail below.

### Getting Setup to Use X

In order to use an X window system for **shape\_edit** graphics commands, the environment variable **DISPLAY** must be set and an X server must be running on that device.

**Shape\_edit** reads default colors, fonts, window positions and window sizes out of the ".Xdefaults" file on the user's home directory. If the user has no ".Xdefaults" file or omits certain entries, system defaults are used. These vary with different hardware, but should do something reasonable in all cases. The text strings for specifying colors are usually found in the file "/usr/lib/rgb.txt" supplied with the X window system. The defaults affecting the XMenu system, e.g., SelectionFont, SelectionStyle, etc., (see "man X" and "man XMenu" for the complete set of options) can also be set. Xgen uses the XMenu package for menus in the GUI and for pop-up help windows. Here is an example for the **shape\_edit** portion of a ".Xdefaults" file:

```
shape_edit.Background:  Black
shape_edit.Border:      Red
shape_edit.Foreground:  White
shape_edit.Depthcue:    LightSteelBlue
shape_edit.ViewForeground: Yellow
shape_edit.ViewBackground: CornflowerBlue
shape_edit.ViewHighlight: SteelBlue
shape_edit.ViewBorder:  White
shape_edit.ViewGeometry: 400x200+0+0
shape_edit.WindowGeometry: 500x500+0+210
shape_edit.WindowStep:   +10+10
shape_edit.Prompts:      500x150-0-0
shape_edit.DEFAULT:      500x500+0+0
```

The parameters have meaning as described below:

#### Background

Sets the background color of **shape\_edit** windows.

#### Border

Sets the border color of **shape\_edit** windows.

#### Foreground

Sets the color of the text and geometric objects.

#### Depthcue

Sets the color of depthcued polylines. (On black and white screens dashed lines are used for depthcuing. This currently does not work on the SUN X server.)

#### ViewForeground

Sets the color of the text in the viewing window.

#### ViewBackground

Sets the color of the background in the main viewing window.

#### ViewHighlight

Sets the color of the subwindows in the viewing window.

#### ViewBorder

Sets the border color for the viewing window and the subwindows.

#### ViewGeometry

Default window position for viewing window.

#### WindowGeometry

A default position for generic **shape\_edit** windows.

#### WindowStep

Relative offset for successive **shape\_edit** windows.

**Prompts** A default position for the prompt window used by the GUI.

**DEFAULT** Any named **shape\_edit** window (DEFAULT in this case) can be given a default position. (Names should be in all upper-case letters.)

### Creating Windows

The mouse can be used to create both **shape\_edit** geometry windows and the viewing window. A prompt string appears in the upper left corner of the screen when a window is being created. Various actions are required of the user, depending on which options are set in the user's ".Xdefaults" file. When a window is completely specified (both position and size) in the ".Xdefaults" file, the window is automatically popped up with no action required from the user. This includes the viewing window, which is created when the xgen device is "grabbed", the "Prompts" window, which is created when the GUI is started, and **shape\_edit** geometry windows, created with the **newWindow** command executed from either the GUI or the programming interface. For example, if the user has the line:

```
shape_edit.FRONTVIEW: 300x300-0+0
```

in the ".Xdefaults" file, the command

```
newWindow( frontView );
```

will automatically pop up the window in the specified position.

For windows that are not completely specified, the mouse is used to create the window. The following describes the actions associated with the mouse buttons:

- Left** This gives a default window, both in size and placement. If **WindowGeometry** is specified in the ".Xdefaults", it is used, otherwise a system default is used. If **WindowStep** is defined in ".Xdefaults", the next window created in this fashion will be offset by **WindowStep**. If the button is held down, the outline of the default window is shown. The mouse can then be dragged to position the window. The window is created when the button is released.
- Middle** This button allows the user to choose the upper left and lower right corners of the window by pressing the middle button to choose the upper left, then dragging the mouse and releasing the button to choose the lower right. An outline is shown while dragging.
- Right** A window of default size (the default is chosen as it is with the left button) is created, but positioned with the upper left corner where the mouse is when the button is released.

### Window Managers

**Shape\_edit** depends on window management (i.e., manipulation of windows after their creation, such as pushing, popping, iconifying, and resizing) to be accomplished by an independent X window manager, such as **wm** (usually running in the background started by a shell). Windows have a minimum size associated with them, and resizing is restricted by the minimum size. Windows which are affected by window manager activity will be redrawn appropriately. Specifically, if the viewing window is resized the subwindows are appropriately scaled.

### The Viewing Window

When xgen is grabbed from **shape\_edit** the viewing window is created. The window acts as a slider

pad and button box emulator to control the display.

The main function of the viewing window is to control the viewing transformations for **shape\_edit** windows. **shape\_edit** windows are initially created with a identity matrix for the viewing transformation and a screen transformation that puts the origin in the middle of the window and sets the scale to be -1 to 1 in the minimum of the X and Y directions. When **shape\_edit** windows are resized their screen transformations are appropriately updated and the windows refreshed.

The sliders in the viewing window are used to change the viewing transformation in a **shape\_edit** window. This can be done at any time. If there is more than one **shape\_edit** window, **xgen** modifies the viewing for the "current" window. This is independent of **shape\_edit**'s notion of the current window. The current window for viewing is marked by appending a ">" to the window name which is displayed in the upper left corner. When multiple windows are used, the current window can be selected by pressing the left mouse button with the cursor in the desired window.

The viewing window consists of many subwindows which emulate slider bars (with either continuous and discrete values) and buttons. The slider bars are used to modify viewing. These subwindows have changing cursors which show which mode the slider is in. The "plus" cursor indicates that only button presses are recognized (discrete values). The "left-right arrow" cursor indicates that the window acts like a continuous slider: A mouse button is pressed and then the mouse is dragged and the button released. In both cases any mouse button can be used, but each mouse button provides a different scale. (See more on modes below.) The left button produces a small scale change the middle button produces a "normal" change and the right button produces a large scale change. In discrete mode, the values are based on the position of the cursor relative to the center of the subwindow.

### Other Viewing Functions

Functions other than changing the viewing transformations are provided by the button emulators. There are four such viewing functions that are always available. They are:

- Reset**      This resets the viewing transformation of the current window to the identity matrix.
- Help**        Pop up a help window. The help window is actually an XMenu "deck of cards" menu with multiple panes. To move from pane to pane, the mouse is moved off the top or bottom of the current pane and the next pane then pops to the top. Clicking any mouse button remove the help window.

### Continuous/Discrete

This sets the mode of the transformation sliders. In continuous mode, the transformation windows act as continuous sliders with the arrow cursor described above. This is useful when displaying small amounts of geometric information in a window, but may be too slow if displaying large amounts of data.

If the **Continuous** box is clicked, the mode changes to **Discrete**. A center line is drawn in each slider and the slider window now uses the plus cursor. Each button press produces a transformation with the signed distance from the center line as a value. Each mouse button provides a different constant multiplier for the value. The left button provides a small constant the constant for the middle button is 1 and the right button provides a large constant.

The text in the **Continuous/Discrete** box displays the current mode.

### More/Less

This option provides a method of adding additional sets of options. When

**More** is selected a menu pops up and the user can choose one of the menu items to add features. Currently there are two such options which are documented with the **help** function (see above). When the extra features are no longer desired (or a different set is desired), the **Less** button can be picked to delete them and return to the original configuration.

### User Interface

A number of options are included to support the GUI. These are fully (and more accurately) documented with the **help** function described above, but described here briefly.

**Menus** When the mouse is in a **shape\_edit** window (or the X root window), clicking the right mouse button pops up the current menu. The menus use the XMenu package and defaults for style and color can be set in the user's ".Xdefaults" file (see above). Menu items are selected by clicking any mouse button.

**Locators** Locator events (generally used for creating points) are generated by clicking the middle mouse button when shifted.

**Pick Events** Objects on the screen can be picked by clicking the middle mouse button (unshifted).

#### Current Window

The current window, for viewing purposes, is selected by clicking the left mouse button in the window. The current window is shown by appending a ">>" to the window name. This does not "select" the window, as far as **shape\_edit** is concerned.

### User Interface Prompting

When the GUI is started (via the `interact();` command) a prompt window is created by `xgen` (see above for specifying a default). This window appears to be a regular **shape\_edit** window with the name "Prompts". As prompts are generated by **shape\_edit** they are displayed in the prompts window, which is automatically scrolled to keep the latest prompts in the window. The text can be scrolled back by treating the window as a graphics window and changing the viewing transformation, but only a finite buffer of prompts is kept. Generally, this is not needed.

Another function of the prompt window is to provide a method for **shape\_edit** to communicate that a long calculation is being done. When this is the case, the message "Please wait" is printed in the upper left corner of the prompt window.

The third function of the prompt window is to allow a primitive method of entering text strings for the various GUI commands that require names or numbers. No quotes are required on names and simple editing is provided by "DELETE", "BACKSPACE", "~U", and "~C" (use **help** function for details).

#### Slider Bar

A slider bar (valuator) is provided for generating numbers for the GUI. It is selected by one of the options on the "More" menu described above. Its use is fully described by the **help** function.

## C.3 Evans & Sutherland PS300 Display

The Evans & Sutherland PS300 graphics display is accessed in **shape\_edit** using the device name "ps300" as in

```
grab ps300;
```

or

```
drop ps300;
```

There are currently no window management facilities on the PS300. The current `shape_edit` window is the only one visible, and it covers the whole screen area. When you grab the PS300 device, the knobs will be loaded and labeled with viewing transformations which can be used at any time to manipulate the view in the current window.

Several function buttons on the keyboard are also useful. The rightmost button toggles between perspective and orthographic display. The leftmost button will reset the viewing matrix to its original values. The second and third buttons on the left can be used to save and restore a particular set of viewing parameters.

## C.4 Evans & Sutherland MPS Display

The Evans & Sutherland MPS (Multi-Picture System) display is accessed from `shape_edit` using the device name "mps" as in

```
grab mps;
```

or

```
drop mps;
```

There are currently no window management functions for the MPS in connection with `shape_edit`, although it is possible to place several windows on the screen. When a window is created, you must use the tablet puck to position the window by holding down any button starting at the upper left corner of the window, dragging to the lower right corner of the window, and then releasing the button. Once the window has been created, it cannot be repositioned (except by dropping and re-grabbing the device, in which case you must reset all the window positions).

A row of "buttons" appears on the screen when you are using the MPS. They are completely non-functional.

The knob box on the mps can be used to change the viewing parameters for the current window at any time. The top 3 knobs on the left are for translations in X, Y, and Z. The top 3 knobs on the right are for rotations about the X, Y, and Z axes. The lower left knob is for global scale.

## C.5 Chromatics

The Chromatics display is accessed from `shape_edit` using the device name "cx3d" as in

```
grab cx3d;
```

or

```
drop cx3d;
```

There is no window manager currently available. Only the current window is visible and it covers the full screen area. The knobs are available for manipulating the view at any time. They include the standard three translations, global scaling, and three rotations about the axes.

This device is not as well debugged as many of the other devices, and it sometimes becomes confused. At Utah, if things go wrong, the following procedure may help:

```
getcx3d $img/models/spoon.rle
```

This will let you know if the Chromatics is talking at all. If it doesn't work, then reboot and try again. Then

```
~wtm/FIX
```

will set the Chromatics to a state which allows **shape\_edit** to work. Then try to grab it from **shape\_edit**.



## Appendix D. Text File Format

This appendix describes the Alpha\_1 text file format for communicating geometric data between programs. This data will almost always be created by programs, but it is sometimes useful to be able to adjust parameters or check the contents of a data file and so some understanding of the text format may be necessary.

A textual input file is a stream of "objects". Certain objects may be imbedded in other objects, e.g., a color in a surface. These objects are sometimes referred to as "attributes".

Comments in the usual C form (`/* This is a comment. */`) may appear anywhere in the file. Any amount of white space may also be used.

An object always has the form:

**keyword** [**instance-name**] = **object-body**

The keyword defines the type of the object being described. The optional instance-name is a string (alpha-numeric including "." and "-" and beginning with an alphabetic) which can later be used as an object-body to get a copy of the first object. The object-body is always enclosed in curly braces, "{" and "}" unless the body is a single numeric value or an instance-name.

With few exceptions, most fields in object-bodies are separated by white space. The exceptions are:

- the coordinates of points and vectors (as in a `Dsurf ctl_mesh` or a `light_vector`) are separated by commas;
- the size fields in `knot_vector` and `ctl_mesh` are denoted in C array. syntax

There are essentially three kinds of objects which can appear in a text file: visible objects, environment objects, and attribute objects.

Visible objects include vertex, polygon, polyline, line, B-spline curve and surface, box-spline surface, and text string. Other visible objects are the "aggregate" objects (group, shell, and instance) which contain other visible objects. These visible objects are the main objects in the file.

Environment objects set up certain environment values used in processing the visible objects by other programs, most notably rendering. Environment objects include screen and viewing transformations, light vectors, and Blinn shading and distribution parameters. Environment objects may appear anywhere in the input stream, but it makes little sense for them to appear after any visible objects.

Attribute objects may be included textually within most visible objects. They include color, back color, width, resolution, opacity, and user-defined attributes with string, integer and float values. These attributes may appear anywhere in the file, but usually must be referenced within an object in order for the attribute to be associated with the object. The exceptions to this are color, back\_color, width, resolution, and opacity (transparency) values for the render program and several other programs. These may be set at the beginning of the input stream, (and not imbedded in any object) and they will be associated with the objects as they are needed and only where appropriate. Attribute references imbedded within objects generally must occur before or after the required fields, but not mixed with the required fields.

There are several attributes which may only be used inside vertex objects. These are normal, shade, and uv attributes. The user-defined string, float and integer attributes are also allowed as vertex attributes.

Perhaps the best explanation of the text form is with a set of examples.

## Environment Objects

A screen transformation may be given as a four-by-four matrix or as a set of parameters. The only explicit matrix that is likely to be useful is the identity matrix in this case (for times when the coordinates of the object are already transformed into actual device coordinates).

```
screen_trans =
{
    1 0 0 0          /* Note space as separator. */
    0 1 0 0
    0 0 1 0
    0 0 0 1
}

screen_trans =      /* All these parameters are optional, */
{                  /* ordering unimportant. */
    hither = -1      /* Default values. */
    yon = 1
    screen = 1        /* Represents half the screen width. */
    eye = -5
    aspect_ratio = 1.2
    viewport = { 0 512 0 480 }
}
```

A viewing transformation may be given explicitly like the identity screen transformation above, or using fields specifying the concatenation of a sequence of primitive transformations. A viewing transformation stack is maintained. When a view\_trans is encountered, it is multiplied into the top of the stack.

```
view_trans =
{
    /* Any combination of these parameters. */
    rotate_x = 30.0
    rotate_y = 30.0
    rotate_z = 30.0
    translate_x = 0.5
    translate_y = 0.5
    translate_z = 0.5
    translate_xyz = { 1.0, 2.0, 3.0 }
    scale_x = 2.0
    scale_y = 2.0
    scale_z = 2.0
    scale_xyz = { 1.0, 2.0, 3.0 }
    scale_uniform = 2.0
    gen_tran = { 1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1 }
    /* axis_rotate =
    /* {
    /*     axis_code = 1          /* 0, 1, or 2 for x, y, z */
    /*     rotate_vector = { 0, 0, 1 }
    /*     axis_to_vector = 0      /* 0 or 1 */
    /* }
    /* axis_rotate_with =
    /* {
```

```

/*      axis_code = 1
/*      rotate_vector = { 0, 0, 1 }
/*      rotate_with_vector = { 1, 0, 0 }
/*      axis_to_vector = 0
/*      }
*/
}

```

When transformations are used in the context of animation, interpolation values may appear as the values of the rotate, translate and scale keywords. So a viewing transformation might take the form:

```

view_trans =
{
  rotate_x =
  {
    start_time = 0.0
    end_time = 1.0
    time_now = 0.0
    start_value = 0.0
    end_value = 180.0
    kinetic_curve =
      Dcurve name1 =
    {
      ...
    }
    position_curve =
      Dcurve name2 =
    {
      ...
    }
  }
}

```

Objects which are defined purely for reference by other objects (and not to be displayed) are called library\_entry objects, and are often useful in animation:

```

library_entry =
{
  /* Any objects can go here. */
}

```

A light vector has comma-separated coordinates.

```
light_vector = { 0, 1, 0 }
```

For the ray-tracer, a different light source specification is used than for the render program.

```

light_source =
{
  /* Optional parameters. */
  location = 10.0 10.0 100.0
  radius = 0.0
  intensity = 1.0
}

```

These are the default blinn parameters; any not specified assume the default values.

```
blinn_params =
{
    refract_index = 100
    specular = .3
    diffuse = .4
}
```

These are the default distribution parameters.

```
dist_params =
{
    weight = 1.0
    slope = .35
}
```

For the ray tracing program, the shading parameters are contained in a surface\_quality object which is an attribute of the individual polygons or surfaces:

```
surface_quality =
{
    refract_index = 1.0
    specular = 0.0
    diffuse = 0.999
    transmit = 0.0
    reflect = 0.0
    glossy_exponent = 1.0
    glossy_diffusion = 1.0
    surface_color = 1.0 1.0 1.0
    sq_name = 'sq1'
}
```

The ray tracing program also uses texture maps for surfaces if they are desired.

```
surface_texture =
{
    texture_type = 0.0
    texture_name = 'texture1'
    resolution_x = 512.0
    resolution_y = 479.0
    rle_image_file = 'texture1.rle'
    surface_qualities = sq1 sq2
}
```

A render mark indicates that no objects occurring beyond this point in the file will begin before this scanline (screen space). These will normally only be generated by the dsurf program.

```
render = 327
```

### Attribute Objects

These objects, when they occur outside visible objects, may be named so they can later be referenced and associated with visible objects. In any case, when they occur outside any visible objects, many programs will add them to a "global attribute list," which is used as a default list when objects don't have attributes that are needed. Render, for example, when trying to display a surface or polygon, will use a default global color attribute if one is not explicitly given with the surface or

polygon.

```
color yellow = { 255 255 0 }    /* Color has red,green,blue fields. */
back_color red = { 255 0 0 }

width fat = 4.0

resolution coarse = 18.0

opacity very_opaque = .9
```

These special attributes (which we sometimes call user-defined attributes) have values which are a string, a float, or an integer. The significance of the values, to a program which looks for them, must be given by the attribute name, which is the second field in each of these examples. The third field is the optional instance name as we saw before. So the form is:

```
keyword attribute-name [instance-name] = attribute-value
```

String values are strings enclosed in single quotes.

```
string_attr comment hello = 'this is a string with blanks'
string_attr comment goodbye = 'this_is_a_different_string'

float_attr pieceofpie one-sixth = .1666666667

integer_attr timeofday six-oclock = 1800
```

It is also possible to define certain names to be particular types of user-defined attributes using the "define" keyword:

```
define string_attr special_name
special_name = 'foo'

define float_attr stress
stress = 0.86

define integer_attr used_elsewhere
used_elsewhere = 1
```

This is just a syntactic shorthand for cases where you need to use a particular user-defined attribute many times. The first example is completely equivalent to

```
string_attr special_name = 'foo'
```

### Visible Objects

Vertex objects should always be named, because they become visible only when referenced in lines or polygons. If no attributes are being specified, the shorthand form shown for examples B through D is acceptable. Otherwise, there must be a coords field as in the first example.

```
vertex A =
{
  coords = { .5, .5, .5 }
  /* Optional attributes. */
  shade = .8
  normal = { 1, 0, 0 }
  uvparms = { 0.3, 0.2 }
```

```

        color = { 255, 255, 255 }
    }

    vertex B = { .6, .6, .6 }
    vertex C = { .7, .7, .7 }
    vertex D = { .8, .8, .8 }

```

A polygon consists of a set of contours, each of which consists of a set of vertices. If a polygon has only one contour and no attributes mentioned, the short form shown in examples one and two can be used. Otherwise the contour fields must be explicitly identified. Another object, called a polyline, is syntactically identical to a polygon in the object body.

```

polygon one = { A B C D }
polyline two = { { 1, 0, 0 } { 0, 1, 0 } { 0, 0, 1 } }

```

The vertices in a contour can be specified using previously defined vertices, or by including the object-body part of a vertex object in place of a vertex name as in the second and third examples below.

```

polygon =
{
    contour = { A B C }
    contour = { B C D }
    /* Optional attributes. */
    color = { 30 40 255 }
}

polygon =
{
    contour = { { 1, 0, 0 } { 0, 1, 0 } { 0, 0, 1 } }
    opacity = .7
}

polygon =
{
    contour =
    {
        {
            coords = { .3, .4, .5 }
            shade = .7
        }
        {
            coords = { .4, .5, .6 }
            shade = .8
        }
        {
            coords = { .5, .6, .7 }
            shade = .9
        }
    }
    color = yellow
}
/* Instance defined above. */

```

Lines are quite similar to polygons, except that they consist of a pair of endpoints. The shorthand form may again be used if there are no attributes.

```

line = { A B }
line = { { .5, .4, .3 } { .3, .4, .5 } }
line =
{
  endpoints = { { .5, .4, .3 } { .3, .4, .5 } }
  /* Optional attributes. */
  color = { 20 50 80 }
  width = fat      /* Attribute value defined above. */
}

```

Dsurf objects have three required parts: the order of the surface in each parametric direction, a pair of knot vectors and the control mesh. Note how the sizes are specified for the knot vectors and the control mesh. The list of attributes may come before and/or after the required fields, but may not be interspersed with them. An optional specification of the end condition types (open, floating, or periodic) may follow the orders of the surface. In the control mesh, an optional tag at the beginning may specify the type of the points in the mesh, and appropriate conversions will be made. If no tag is given, the point type is determined by the dimension of the first point. In the ambiguous case of three coordinates (which could be E3 or P2) E3 is chosen. The keywords for point types are SCALAR, E2, E3, P2, P3, R2, R3, or RN size where size is the size of the RN vector.

```

Dsurf =
{
  /* Optional attributes. */
  orders = { 3 3 }
  ec_types = { open, open } /* Optional */
  knot_vector[7] = { 0 0 0 0.25 0.5 0.5 0.5 }
  knot_vector[9] = { 0 0 0 1 2 3 4 4 4 }

  /* Coordinates are separated by commas; points are
   * separated by white space. Points may be given as 2D,
   * 3D, or 4D and may be mixed within a control mesh with
   * one restriction: If ANY 4D points occur, the first
   * point must be given as 4D. 2D points are assigned a
   * Z = 0, and 3D points, if they occur in a 4D control
   * mesh are given a W = 1.
   */
  ctl_mesh[6][4] =
  {
    E3 /* Optional point tag. */

    -0.300000, 0.000000, 0.000000
    -0.300000, 0.000000, -0.300000
    0.300000, 0.000000, -0.300000
    0.300000, 0.000000, 0.000000

    -0.500000, 0.375000, 0.000000
    -0.500000, 0.375000, -0.500000
    0.500000, 0.375000, -0.500000
  }
}

```

```

        0.500000, 0.375000, 0.000000
    -0.150000, 0.725000, 0.000000
    -0.150000, 0.725000, -0.150000
    0.150000, 0.725000, -0.150000
    0.150000, 0.725000, 0.000000

    -0.250000, 0.750000, 0.000000
    -0.250000, 0.750000, -0.250000
    0.250000, 0.750000, -0.250000
    0.250000, 0.750000, 0.000000

    -0.250000, 0.800000, 0.000000
    -0.250000, 0.800000, -0.250000
    0.250000, 0.800000, -0.250000
    0.250000, 0.800000, 0.000000

    -0.200000, 0.800000, 0.000000
    -0.200000, 0.800000, -0.200000
    0.200000, 0.800000, -0.200000
    0.200000, 0.800000, 0.000000
}
opacity = 0.200000
resolution = 3.000000
color = { 191 142 57 }
}

```

Curves look very similar to surfaces.

```

Dcurve =
{
    /* Optional attributes. */
    order = 3
    ec_type = floating /* Optional field. */
    knot_vector[7] = { 0 0 0 0.25 0.5 0.5 0.5 }
    ctl_poly[4] =
    {
        E3
        -0.300000, 0.000000, 0.000000
        -0.300000, 0.000000, -0.300000
        0.300000, 0.000000, -0.300000
        0.300000, 0.000000, 0.000000
    }
    /* Optional attributes. */
    color = { 191 142 57 }
    width = 2.0
}

```

A special class of multivariate spline surfaces, called box-splines, may be specified in a text file. Optional attribute fields may precede or follow the required fields. The required fields include the degree of the surface, a knot set consisting of pairs of parametric values, and a control mesh which may contain special "null" points as shown below. You probably have to know a lot about



box-splines to try to define one of these!

```

Msurf =
{
  /* Optional attributes. */
  degree = 3
  knot_set[5] = {
    (1,0)
    (0,1)
    (1,1)
    (1,1)
    (1,1)
  }
  ctl_mesh[4][4] = {
    0, 0, 0
    1, 0, 2
    null_point, 0, 0
    null_point, 0, 0

    0, 1, 2
    1, 1, 2
    2, 1, 2
    null_point, 0, 0

    null_point, 0, 0
    1, 2, 2
    2, 2, 2
    3, 2, 0

    null_point, 0, 0
    null_point, 0, 0
    2, 3, 0
    3, 3, 0
  }
  /* Optional attributes. */
  color = {0 75 75}
  resolution = 1.0
}

```

A group object is used to collect several other objects (possibly including other groups) into a single higher-level structure. It provides a convenient structuring mechanism for complex models.

```

group =
{
  /* Optional attributes. */
  members =
  {
    line = { A B }
    polyline = { { 0, 1, 0 } { 0, 0, 0 } { 0, 0, 1 } }
  }
  /* Optional attributes. */
}

```

```

}
```

An instance object is just some other object coupled with a transformation which indicates a new position for the geometry than the coordinates which were used in the definition of the object.

```

instance =
{
    /* Optional attributes. */
    instance_obj =
    {
        Dcurve =
        {
            order = 2
            knot_vector[4] = { 0 0 1 1 }
            ctl_poly[2] = { 0, 0    0, 1 }
        }
    }
    instance_trans = /* Can include anything like a view_trans. */
    {
        rotate_x = 30
        translate_xyz = { 2, 3, 4 }
    }
    /* Optional attributes. */
    back_color = red
}
}
```

A shell object is used to collect several surfaces into a single entity for use with the combiner program. Unlike groups, only surface objects may appear in the shell.

```

shell =
{
    /* Optional attributes. */
    shell_surfs =
    {
        Dsurf =
        {
            orders = { 2 2 }
            knot_vector[4] = { 0 0 1 1 }
            knot_vector[4] = { 0 0 1 1 }
            ctl_mesh[2][2] =
            {
                0, 0
                0, 1
                1, 0
                1, 1
            }
        }
        Dsurf =
        {
            orders = { 2 2 }
            knot_vector[4] = { 0 0 1 1 }
            knot_vector[4] = { 0 0 1 1 }
            ctl_mesh[2][2] =

```

```

        {
            0, 0
            0, -1
            -1, 0
            -1, -1
        }
    }
    /* Optional attributes. */
}

```

Set expressions are instructions for the combiner program to use when trimming one shell against another. Set union, intersection, difference and negation can be combined in a single expression which dictates exactly which portions of which surfaces are not part of the model.

```
set_expr = { ( foo + bar ) - ~fee * fum }
```

Text string objects have two possible forms. In the second form, optional attributes may appear before or after the two required fields.

```

text_string = { 'This is my string' } text_string = {
    /* Optional attributes. */
    text = 'This is my string'
    position = { 0.1, 0.7 }
    /* Optional attributes. */
}

```

### Adjacencies

Adjacencies can be declared between various objects. These are actually attribute objects, but are always generated by programs. There may be multiple "edge\_adjacency" fields to specify the adjacencies to the various edges of the object and multiple "e\_adj\_part" fields to specify multiple edges of other object adjacent to a single edge of the current object.

```

adjacency =
{
    edge_adjacency =
    {
        parent_name = 'S0'
        parent_edge = 4
        e_adj_part =
        {
            adj_name = 'P8'
            adj_edge = 3
            parent_range = { { 1.0 1.0 } { 0.0 1.0 } }
            adj_range = { { 1.0 1.0 } { 0.0 1.0 } }
        }
    }
}

```



## Appendix E. Rlisp Syntax Summary

This appendix provides a brief, informal description of Rlisp syntax, although the examples in the main text of this manual should be adequate for beginning users of the Alpha\_1 system. A more formal discussion of Rlisp may be found in Part 2 of the PSL User's Manual.

Rlisp is simply a surface language which allows the underlying Lisp system to be communicated with in a manner which is similar to many standard programming languages such as C, Pascal, or Algol. Some familiarity with programming concepts and data structures is assumed for this appendix.

### Comments

Comments in Rlisp begin with a “%” character and continue to the end of the line on which they begin.

```
% This is a comment.
```

### Separators

Statements in Rlisp are usually separated by the “;” character. Most of the time, you can get away with thinking of them as terminators for statements, as in many other languages. But there are a few cases where you need to be conscious of the difference. The most important is within an “if-else” statement. There must not be a semicolon before the word “else”. The other cases are related to returning values from functions: if there is a semicolon before the closing brace and you are not using a “return” statement, then the value Nil will be returned, regardless of what computed value was the result of the last statement.

The “\$” character is equivalent to “;” from a syntactic point of view, and can be used anywhere the “;” character is used. The difference is in the output which Rlisp prints as the result of evaluating an expression. Normally the result is printed to your screen, but if the expression results in a large object (like a surface) you may be more interested in looking at a graphical display than a textual representation. The “\$” used as a separator causes the textual output to be suppressed, and the next prompt is just printed without showing the value of the last expression.

### Assignment Statements, Variables, Constants

In Rlisp, a variable X may be assigned a value 1 using the “:=” operator. In `shape_edit`, a special variant “^=” of the assignment operator is defined. This version allows the results of a computation to be displayed on the graphics screen as part of the assignment statement. (Constructing an object and displaying it graphically is a very common sequence in `shape_edit`, and it is convenient to be able to say it in one line rather than two.)

```
X := 1;                                % Assignment statement.
```

In this case the constant integer value 1 was assigned to X. Some other constants are given by:

```
X := 1.0;                             % Floating point constant.
X := "String";                         % String constant.
X := '( 1 2 3 );                      % List constant, with three integer
                                     %   elements.
X := '[ 1 2 3 ];                      % Vector constant, with three
                                     %   integer elements.
X := 'Identifier;                     % An id.
```

Variables in Rlisp are not typed, so any kind of value can be assigned to any variable. Variable names are strings of characters which may include alphabetic and numeric characters as well as certain others. They can be very long.

Individual elements of a variable which has a vector value (or a list value actually) may be accessed using an index:

```
ValueX := VectorX[3];
```

### Arithmetic and Logical Operators

All of the operators below are used in infix form (i.e., A Op B) in Rlisp. They are listed in order of precedence, with equal precedence operators grouped together:

<b>**</b>	Exponentiation
<b>/</b>	Division
<b>*</b>	Multiplication
<b>-</b>	Subtraction
<b>+</b>	Addition
<b>=</b>	Equal to
<b>&gt;=</b>	Greater than or equal to
<b>&gt;</b>	Greater than
<b>&lt;=</b>	Less than or equal to
<b>&lt;</b>	Less than
<b>Neq</b>	Not Equal
<b>And</b>	Logical and
<b>Or</b>	Logical or

### Procedure Calls

An Rlisp procedure which has already been defined is invoked as:

```
procedureName( Arg1, Arg2, ..., ArgN );
```

If the procedure has no arguments, then:

```
procedureName();
```

If it has just one argument, then the following two forms are equivalent:

```
procedureName( Arg );
procedureName Arg;
```

### Defining Procedures

A new procedure may be defined and executed at any time during the Rlisp session. The simplest kind of procedure is defined as:

```
procedure myProcedure( Arg1, Arg2 );
  Arg1 + Arg2;
```

The "procedure" keyword is followed by the name of the new procedure and its arguments (just like in the procedure call). The body of the procedure is a single expression, whose value is the result of the procedure call (all Rlisp procedures return some value).

A slightly more complicated example involving local variables is:

```
procedure myProcedure( Arg1, Arg2 );
begin
  scalar ReturnValue;
```

```

    ReturnValue := Arg1 + Arg2;
    return ReturnValue;
end;
```

In this case the “begin” and “end” form a block which may contain multiple statements. The “return” statement is used to specify the resulting value of the routine. The “scalar” statement declares local variables and must occur immediate following the “begin”.

Blocks can also be used to group statements within a procedure, but within a procedure, the shorthand symbols “{” and “}” may be used to delimit the block.

Alpha\_1 convention is that procedure names begin with a lower case letter, with beginnings of imbedded words capitalized (e.g., **arcRadTan2Lines**). All variables and formal parameters to procedures begin with a capital letter and follow the same convention for imbedded words. Names should be as long as necessary to be mnemonic.

### If Statements

The form of an if-then-else statement is one of:

```

    if condition then expression1 else expression2;
    if condition then expression1;
```

Note that in the first form there is no semi-colon following the expression. A correct statement might be:

```

    if ValueA > ValueB then
        X := ValueA
    else
        X := ValueB;
```

Blocks can be used for the expressions:

```

    if ValueA > ValueB then
    {
        ...
    }
    else
    {
        ...
    };
```

### Looping Constructs

A simple iteration loop which steps the value of a variable is:

```

    for I := Begin : End do                % Step from Begin to End by 1
    {
        ...
    };
```

A slightly more general form is:

```

    for I := Begin step Increment until End do    % Step by Increment
    {
        ...
    };
```

Standard while and repeat loops are also provided:

```

while ValueA < ValueB do
{
    ...
};

repeat
{
    ...
} until ValueA < ValueB;

```

Since lists of objects are common in Rlisp, it is often useful to perform a sequence of actions on a list of objects. A special looping construct for this purpose is:

```

foreach Object in ObjectList do
{
    ...
};

```

The value of `Object` will be set to each successive object in the list as the loop is executed.

One can form a list of the results of executing such a loop by using the "collect" keyword instead of "do", as in:

```

ResultList := foreach Object in ObjectList collect
{
    ...
};

```

The `ResultList` will be a list of the values returned for each iteration through the loop. The "collect" keyword can also be used with the "for I = Begin : End" form.

### Accessing Objects

Most of the datatypes you will encounter in `shape_edit` are actually objects. These are a special kind of data structure with a number of properties that are used extensively in `shape_edit`. Fields of an object are accessed by a procedure which has the same name as the field. The C-like ">" syntax is often used instead for accessing fields of objects. The following three statements are all equivalent:

```

cPoly( NewSrf );
cPoly NewSrf;
NewSrf->cPoly;

```

**Reading Other Files** There are two commands in Rlisp for reading other files. The `in` command is for reading ".r" files (text sources), and the `load` command is for reading ".b" files which have been compiled.



## Appendix F. Gemacs & Shape\_edit

The gemacs Rlisp mode includes the ability to fire up any of the PSL Rlisp as a subprocess of gemacs, feeding them input from any rlisp-mode buffer in the gemacs on a statement at a time basis for nice interaction.

The gemacs function `rlisp-execute`, bound to “Meta-e”, starts the desired rlisp process, specified by variable `rlisp-pgm`. Alpha\_1-ers will want to start up `shape_edit`, and thus the standard gemacs startup file “`alpha1.ml`” contains:

```
(setq-default rlisp-pgm "~alpha1/bin/shape_edit")
```

A buffer named “`rlisp`” receives the output from the process, and is automatically popped up on the gemacs screen. To kill the rlisp process, switch over to that window and hit “`^\\`” (control-backslash.) Then the next input via “Meta-e” will start it up again.

“`^U` Meta-e” suppresses the echo of the input to the rlisp process in the rlisp buffer, which is nice when long function definitions are being fed in. Either form of “Meta-e” continues feeding input until it reaches the end of the Rlisp statement, or the end of the input buffer.

“`^Z-s`” prompts for a single line and sends it.

“`^Z-b`” is a prefix for sending single-character break loop commands to Rlisp. For example “`^Z-b-q`” is `breakQuit`. “`^Z`” followed by a digit, “`y`” or “`n`” is good for answering trace command or function redefinition queries. Either command can be executed from any window on the screen.



## Index

\$	311	all edges menu item	287
%	311	ALLedges	90
*	312	allstate	215
**	312	alphaout	219, 227
+	312	ambient_light attribute	237
-	312	and	312
->	314	angDimAttr	159, 272
.Xdefaults file	293	AngDimDefaultAttrs	158
/	312	angle between menu item	278
:=	35, 311	angleFrom2Planes	60, 257
;	311	angleOfLine	54, 255
=	312	anything datatype	25
[ ]	42, 312	anyVector datatype	25
{ }	313	append	42, 251
>	312	appendDescriptor	147, 149, 270
>=	312	approximate values menu item	288
^=	35, 311	approxValuesSplineFn	126, 266
<	312	arc datatype	25
<=	312	arc end menu item	279
2 circles intersect menu item	279	arc pick menu item	278
2 circles menu item	281	arc start menu item	279
2 lines intersect menu item	279	arcCutFromCircle	67, 258
2 points menu item	280, 281	arcEnd	65, 258
2 pt distance menu item	278	arcEndCenterEnd	63, 257
2 pt extrude menu item	282	arcEndTan2Lines	63, 257
2 pt interp menu item	279	arcRadTanToCircleAndLine	64, 258
2 vec interp menu item	280	arcRadTan2Circles	67, 258
2 vector perp menu item	280	arcRadTan2Lines	63, 257
2DLine datatype	25	arcs	61
2DPt datatype	25	arcs & circles menu	275
3 lines menu item	281	arcStart	65, 258
3 planes menu item	279	arcTan3Lines	63, 257
3 points menu item	281	arcThru3Pts	63, 257
3DLine datatype	25	arender	220, 227
3DPt datatype	25	aspect	220, 227
4 boundary curves menu item	283	aspect ratios	205
<b>A</b>		at angle menu item	280
a1 files	299	attr datatype	26
add constant menu item	279	attribute	181, 182, 270
add flex menu item	284	attributeName	182, 270
add to group menu item	285	attributeValue	182, 270
add vectors menu item	280	autoMkAdjacent	183, 271
addFlex	116, 264	autoMkAdjacentL	184, 271
addFlexObj	264	axis	246
addRestriction	134, 265	axis curve	99
addToGroup	150, 267	axis to vector menu item	285
adj	229	axis to vector orient menu item	285
adjacency	309	alps	241
adjacency attributes	183	alps program	241
adjacency declarations	191	alps_defaults	242
adoptDimDefaultAttrs	160, 272	alps_state	242
aggregates menu	275	alps2D	242
		alps3D	242

altose . . . . . 185

## B

B-spline curve . . . . . 11  
 B-splines in Alpha\_1 . . . . . 4  
 back color parameter . . . . . 222  
 back face menu item . . . . . 287  
 back\_color . . . . . 303  
 background . . . . . 208, 218, 226  
 basic curves menu . . . . . 275  
 basic surfaces menu . . . . . 275  
 basis functions . . . . . 11  
 bBox datatype . . . . . 27  
 bbox max menu item . . . . . 279  
 bbox min menu item . . . . . 279  
 bboxObj . . . . . 43, 271  
 begin statement . . . . . 313  
 bend . . . . . 122, 264  
 bend menu item . . . . . 284  
 bezier menu item . . . . . 287  
 BFACE . . . . . 89  
 bindModelToSymtab . . . . . 186, 271  
 blended sweeps . . . . . 100  
 blending functions . . . . . 11  
 blinn . . . . . 216, 226  
 blinn lighting model . . . . . 224  
 blinn\_params . . . . . 224, 302  
 blinnParam . . . . . 224, 271  
 boolean datatype . . . . . 24  
 boolean operations . . . . . 191  
 boolean sum . . . . . 98  
 boolSum . . . . . 98, 264  
 bottom menu item . . . . . 286  
 boundary representation . . . . . 3  
 bounded flat menu item . . . . . 282  
 bounding box . . . . . 43  
 bounds . . . . . 231  
 box . . . . . 85, 263  
 box datatype . . . . . 26  
 box with planes menu item . . . . . 283  
 box with vecs menu item . . . . . 283  
 brl\_aspect . . . . . 220, 227  
 buffer . . . . . 219, 227  
 buildRestriction . . . . . 134, 264  
 built-in attributes . . . . . 182

## C

cAddFlex . . . . . 137, 266  
 calc program . . . . . 245, 246  
 calc\_norms . . . . . 208  
 calcdetails . . . . . 246  
 calclip . . . . . 246  
 calcstate . . . . . 246  
 camera datatype . . . . . 27  
 Cartesian vector . . . . . 44

center . . . . . 206  
 center of menu item . . . . . 279  
 center point menu item . . . . . 281  
 center radius menu item . . . . . 281  
 centerOfArc . . . . . 65, 258  
 cFeatureLine . . . . . 138, 266  
 changeDimAttr . . . . . 160, 272  
 changeDimDefaultAttr . . . . . 160, 272  
 checkRestrictions . . . . . 134, 265  
 chord menu item . . . . . 287  
 ChordLength . . . . . 92  
 chordlength menu item . . . . . 286  
 circle datatype . . . . . 25  
 circle pick menu item . . . . . 278  
 circleCtr . . . . . 66, 258  
 circleCtrPt . . . . . 66, 258  
 circleCtrRad . . . . . 66, 258  
 circleCtrRadNormal . . . . . 66, 258  
 circleRad . . . . . 66, 258  
 circleRadTan2Circles . . . . . 66, 258  
 circTubeConstantWidth . . . . . 102, 265  
 circTubeWithProfile . . . . . 102, 265  
 circular arcs . . . . . 61  
 cIsolateRegion . . . . . 138, 266  
 cKv . . . . . 74, 260  
 cKvType . . . . . 74, 260  
 clear window menu item . . . . . 277  
 clearDev . . . . . 32, 250  
 clearDevWindow . . . . . 34, 251  
 clearWindow . . . . . 34, 251  
 cLift . . . . . 138, 266  
 CLPathName!\* . . . . . 176  
 cMeshSize . . . . . 259  
 coercion functions . . . . . 45  
 coercion functions for lines . . . . . 53  
 COL . . . . . 75  
 collect statement . . . . . 314  
 color . . . . . 303  
 color parameter . . . . . 222  
 colors . . . . . 204  
 colParminfo . . . . . 78  
 colParmInfo . . . . . 261  
 column menu item . . . . . 286  
 comb\_eps . . . . . 198  
 comb\_lines . . . . . 198  
 comb\_nolines . . . . . 198  
 comb\_normal . . . . . 198  
 comb\_state . . . . . 198  
 combine program . . . . . 197  
 combinerObj datatype . . . . . 26  
 combinerObject . . . . . 197, 267  
 combining objects . . . . . 191  
 command interface . . . . . 9  
 command-driven interface . . . . . 20  
 comments . . . . . 311

complete cubic menu item . . . . . 284  
 completeCubicInterp . . . . . 92, 267  
 computeNodes . . . . . 75, 261  
 cone datatype . . . . . 26  
 cones . . . . . 86  
 constant menu item . . . . . 288  
 constantSplineFn . . . . . 126, 266  
 construction of a shell . . . . . 191  
 constructive solid geometry . . . . . 3  
 continuity . . . . . 12  
 control mesh . . . . . 15  
 control points . . . . . 11  
 control polygon . . . . . 11  
 conv program . . . . . 189  
 convex hull property . . . . . 5  
 Coons' patches . . . . . 98  
 coordinate systems . . . . . 29  
 coordinates menu item . . . . . 279, 280  
 copyDevWindow . . . . . 33, 251  
 cOrder . . . . . 74, 260  
 cornell . . . . . 216, 226  
 cornell lighting model . . . . . 224  
 cosine . . . . . 216, 226  
 cosine lighting model . . . . . 224  
 counterBore . . . . . 164, 272  
 counterDrill . . . . . 164, 272  
 counterSink . . . . . 164, 272  
 cParmInfo . . . . . 74, 260  
 cPoly . . . . . 74, 260  
 cPolySize . . . . . 259  
 create parminfo menu item . . . . . 287  
 cRefine . . . . . 72, 260  
 cross section curve . . . . . 99, 102  
 crossProd . . . . . 50, 254  
 crvData datatype . . . . . 27  
 crvDerivEval . . . . . 73, 260  
 crvEval . . . . . 73, 260  
 CrvFineness . . . . . 37  
 crvfineness . . . . . 206, 242  
 crvFromArc . . . . . 68, 260  
 crvFromCircle . . . . . 68, 260  
 crvFromCrvProjOntoPlane . . . . . 61, 257  
 crvFromParmInfoAndPositions . . . . . 93, 267  
 crvFromSrf . . . . . 78, 262  
 crvInSrf . . . . . 79, 262  
 crvNthDerivEval . . . . . 74, 260  
 CrvPolys . . . . . 36  
 crvPt . . . . . 75, 260  
 crvTangLines . . . . . 137, 266  
 crvTangLines2 . . . . . 137, 266  
 CSG . . . . . 3  
 cshade program . . . . . 207  
 ctlMesh . . . . . 259  
 ctlMesh datatype . . . . . 25  
 ctlMeshTranspose . . . . . 259

ctlPoly . . . . . 259  
 ctlPoly datatype . . . . . 25  
 cType . . . . . 74, 260  
 CUBIC . . . . . 71  
 cubic menu item . . . . . 284, 287  
 cubicCrvFromParametersAndPositions . . . . . 93, 267  
 cull . . . . . 218, 227  
 curve . . . . . 67, 68, 260  
 curve datatype . . . . . 25  
 curve derivative menu item . . . . . 280  
 curve eval menu item . . . . . 279  
 curve fineness menu item . . . . . 277  
 curve fitting menu . . . . . 275  
 curve menu item . . . . . 282  
 curve n derivative menu item . . . . . 280  
 curve offsets . . . . . 139  
 curve pick menu item . . . . . 278  
 curve polys menu item . . . . . 277  
 curve refinement . . . . . 72  
 curve region menu item . . . . . 282  
 curveOpen . . . . . 72, 261  
 cut from circle menu item . . . . . 281  
 cvOffset . . . . . 138, 266  
 cx3d device . . . . . 297  
 cylinder datatype . . . . . 26  
 cylinder menu item . . . . . 283  
 cylinders . . . . . 86

## D

dataItem datatype . . . . . 27  
 DBedge . . . . . 90  
 dbg . . . . . 231  
 dcurve . . . . . 306  
 deconv program . . . . . 190  
 defaspect . . . . . 220, 227  
 defaults . . . . . 216  
 defcrvfineness . . . . . 206, 242  
 define . . . . . 303  
 defining procedures . . . . . 312  
 deflxcolors . . . . . 234  
 deflxlevels . . . . . 234  
 defParamType . . . . . 153, 271  
 defprolog . . . . . 242  
 defpsheight . . . . . 241  
 defquickaspect . . . . . 235  
 defraybuff . . . . . 237  
 defraygrid . . . . . 238  
 defrayheight . . . . . 239  
 defraysample . . . . . 238  
 defraysize . . . . . 239  
 defraythresh . . . . . 238  
 defraywidth . . . . . 239  
 defsrffineness . . . . . 206, 242  
 deftessres . . . . . 230  
 defxraymag . . . . . 240

defxres	220, 227
defyres	220, 227
degree raising	72, 79
delete from group menu item	285
deleteDescriptor	148, 149, 270
deleteFromGroup	150, 267
density	246
dependency propagation	19, 39
depositor functions	45
depsilon	231
describe object menu item	276
deselect window menu item	277
deselectAll	33, 251
deselectDevWindow	33, 251
deselectWindow	33, 251
DFACE	89
DFedge	90
diaDimAttr	159, 272
DiaDimDefaultAttrs	158
diffCrv	73, 260
diffGeom datatype	27
diffSrf	80, 262
dimensioning	157
direction extrude menu item	282
directionOfArc	65, 258
dirOfLine	54, 255
dirPerpLine	54, 255
dirPerpLineInPlane	54, 255
disjoint attribute	201
DISPLAY environment variable	293
display utilities	229
dist_params	225, 302
distLineLine	56, 255
distParam	225, 271
distPtLine	56, 255
distPtPt	48, 254
divide menu item	279
do statement	313
dotProd	50, 254
dottedPairOf datatype	24
down face menu item	287
drop	31, 250
dsurf	229, 305
dump to file menu item	276
dumpA1File	185, 271
dynamic	217, 226

**E**

e2	45, 53
EC_FLOATING	71
EC_OPEN	71
EC_PERIODIC	71
edge between menu item	287
electricConnectorSet	165, 272
ellipsoid	87, 263

ellipsoid datatype	26
else statement	313
emacs text editor	9
end center end menu item	281
end conditions	11
end statement	313
endDirOfArc	65, 258
epsilon	231
Euclidean point	44
euclideanP	46, 253
euclidPoint datatype	25
exponential RLE format	237
extractMatrix	147, 149, 270
extractor functions	44
extrude	83, 264
extrudeDir	83, 264
extrusion	83
e2	253, 256
e2Pt datatype	25
e3	45, 53, 253, 256
e3Pt datatype	25

**F**

fasthighlights	217, 226
feature line menu item	284
featureLine	122, 264
FFACE	89
filter	220, 227
finite element analysis	177
first	42, 252
fitSpec datatype	27
fitting keywords menu	275
flat	217, 226
flat menu item	282
flat surfaces	96
flatSrf	96, 264
flatSrfBounds	98, 264
flatten menu item	285
flattenSrf	132, 264
flattenSrfR	133, 264
flip	217, 226
float datatype	24
float_attr	303
floatAttr	182, 270
floating end condition	11
floating menu item	287
flush	32, 250
flushAll	32, 250
font attribute	243
font_size attribute	243
for statement	313
foreach statement	314
forget object menu item	276
fourperflat	219, 227
fourth	42, 252

frenet frame . . . . .	139
from arc menu item . . . . .	282
from circle menu item . . . . .	282
from curve menu item . . . . .	287
from surface mesh menu item . . . . .	282
from surface u (row) dir menu item . . . . .	287
from surface v (col) dir menu item . . . . .	287
front face menu item . . . . .	287
function datatype . . . . .	24

## G

general sweeps . . . . .	99
generalSweep . . . . .	104, 265
generic methods . . . . .	154
generic operations . . . . .	43
genTran . . . . .	268
geometric editor . . . . .	7
geometric information . . . . .	69
geometry field of primitives . . . . .	86
geomVector datatype . . . . .	25
get300mat program . . . . .	207
getAdjacencyVectors . . . . .	184, 250
getAttr . . . . .	181, 270
getbob program . . . . .	215
getBoundary . . . . .	78, 262
getcmds program . . . . .	189
getCrvFineness . . . . .	37, 250
getCrvPolys . . . . .	37, 249
getcx3d program . . . . .	215
getfb . . . . .	219
getfb program . . . . .	215
getiris program . . . . .	215
getIsoLines . . . . .	38, 250
getmex program . . . . .	215
getReverseNorms . . . . .	38, 250
getSmoothCrvs . . . . .	37, 249
getSrfFineness . . . . .	38, 250
getSrfMeshes . . . . .	37, 250
getSrfNorms . . . . .	38, 250
getX program . . . . .	215
global attributes . . . . .	181
goals of Alpha_1 . . . . .	4
goodhighlights . . . . .	217, 226
grab . . . . .	31, 250
graphical dimensions . . . . .	157
graphical transformations . . . . .	43
graphical user interface . . . . .	9, 20
graphics in Alpha_1 . . . . .	6
gray_level attribute . . . . .	243
group . . . . .	150, 267, 307
group datatype . . . . .	26
group menu item . . . . .	285
GUI . . . . .	9, 20

## H

hardcopy . . . . .	241
hiddefaults . . . . .	232
hidden program . . . . .	231
hidstate . . . . .	232
highLight . . . . .	36, 249
highlight object menu item . . . . .	276
horizontal menu item . . . . .	280

## I

if statement . . . . .	313
in surface menu item . . . . .	282
indexDescriptor . . . . .	148, 149, 270
init method . . . . .	155
insertDescriptor . . . . .	147, 149, 270
inside . . . . .	29
instance . . . . .	149, 267, 308
instance datatype . . . . .	26
instance menu item . . . . .	285
intAttr . . . . .	182, 270
integer datatype . . . . .	24
integer_attr . . . . .	303
interact . . . . .	31, 250
interpolate values menu item . . . . .	288
interpVal datatype . . . . .	27
interpValuesSplineFn . . . . .	127, 266
inverseCounterBore . . . . .	164, 272
invObjTransform . . . . .	149, 267
iris device . . . . .	289
iso . . . . .	206, 242
isolate region menu item . . . . .	284
isolateRegion . . . . .	118, 264
IsoLines . . . . .	37
isoparametric line . . . . .	17
isoparametric menu item . . . . .	277

## J

join cubic menu item . . . . .	284
join linear menu item . . . . .	284
join quadratic menu item . . . . .	284

## K

keystrokes in GUI . . . . .	22
keyword datatype . . . . .	24
keywords menu . . . . .	275
knot vector . . . . .	11
knotList datatype . . . . .	25
knots . . . . .	11
knotVector . . . . .	259
knotVector datatype . . . . .	25
KV_BEZIER . . . . .	71
KV_CHORD . . . . .	71
KV_UNIFORM . . . . .	71
kvNormalize . . . . .	259
kvSize . . . . .	259

**L**

LBedge	90
LDedge	90
left face menu item	286
left menu item	286
left mouse button in GUI	22
lexview program	233
LFACE	89
LFedge	90
libraries	150
library entry	150
library_entry	301
libraryEntry	151, 267
lift	113, 264
lift menu item	284
light sources	223
light_source	301
light_vector	301
lightSource	237, 271
linDimAttr	160, 272
LinDimDefaultAttrs	158
line	305
line datatype	25
line direction menu item	280
line pick menu item	277
LINEAR	71
linear menu item	287, 288
linear sweeping	83
linearPattern	165, 272
linearSplineFn	126, 266
lineHorizontal	51, 255
lineIntersect2Planes	53, 255
lineOffsetFromLine	52, 254
lineP	54, 256
linePtAngle	52, 254
linePtCircle	53, 254
linePtParallel	52, 254
linePtVec	52, 254
lines	51
lines & planes menu	275
linesParallelP	54, 256
linesSkewP	54, 256
lineTan2Circles	53, 255
lineThru2Pts	52, 254
lineVertical	51, 255
list	41, 251
listOf datatype	24
local control	5
loftHit	135, 264
loops in combiner	199
LUedge	90
lxcolors	234
lxlevels	234
lxmeshes	234
lxquick	234

lxrender program	233
lxverbose	234

**M**

make adjacent menu item	285
make transform menu item	286
makeDrill	167, 272
makeEndMill	166, 272
makeNcPosition	172, 273
makeNcStateVec	169, 273
makeTap	167
MakeTap	272
makeToolPosition	168, 272
matDescr datatype	26
matrix descriptor	143
matrix_aspect	220, 227
matrix4x4 datatype	26
memberOfGroup	150, 267
menu-driven interface	20
mergeKv	259
mesh	231
meshes	206, 242
metallic rendering	225
method procedures	154
middle mouse button in GUI	22
mkAdjacent	183, 270
mkCompatible	73, 261
model	39
modify object menu item	276
modify point menu item	276
mouse buttons in GUI	22
moveDevWindow	33, 251
mps device	297
msurf	307
multiple knot	11
MultiWindow	33
mung program	205, 212

**N**

name object menu item	276
narrow	241
NC machining	166
NcBlockInc!*	170
ncChordParms	173, 273
ncCmds2d	175
ncControl	171, 273
ncCopyState	170, 273
ncCrvMill	174, 273
ncFile	171, 273
ncGen	171, 273
ncGenProfile	174, 273
ncNewTape	170, 273
ncProfileOffset	176, 273
NcRounding!*	170
ncSafeBlock	171, 273



NcSafeZ!\* . . . . . 173  
 NcSlowFeed!\* . . . . . 173  
 ncSrfIso . . . . . 172, 273  
 ncSrfIsoZag . . . . . 172, 273  
 ncStateInit . . . . . 171, 273  
 neatcorners . . . . . 218  
 negate menu item . . . . . 279  
 neq . . . . . 312  
 new window menu item . . . . . 277  
 newCMesh . . . . . 259  
 newCPolygon . . . . . 259  
 newDevWindow . . . . . 32, 251  
 newKnotVector . . . . . 259  
 newPolygon . . . . . 256  
 newPolyline . . . . . 256  
 newWindow . . . . . 32, 251  
 no menu item . . . . . 286  
 noadj . . . . . 229  
 noalphaout . . . . . 219, 227  
 noarender . . . . . 220, 227  
 noaxis . . . . . 246  
 nobackground . . . . . 208, 218, 226  
 nobounds . . . . . 231  
 nobuffer . . . . . 219, 227  
 nocalc\_norms . . . . . 208  
 nocalcflip . . . . . 246  
 nocenter . . . . . 206  
 nocull . . . . . 218, 226  
 nodbg . . . . . 231  
 nofilter . . . . . 220, 227  
 noflip . . . . . 217, 226  
 noise . . . . . 206, 242  
 nolxmeshes . . . . . 234  
 nolxquick . . . . . 234  
 nolxverbose . . . . . 234  
 nomesh . . . . . 231  
 nomeshes . . . . . 206, 242  
 nonarrow . . . . . 241  
 noneatcorners . . . . . 218  
 noorigin . . . . . 246  
 nopolys . . . . . 206, 242  
 nopseudocolor . . . . . 217, 226  
 noquickclear . . . . . 235  
 noquickclip . . . . . 235  
 noquickoverlay . . . . . 235  
 noquicktran . . . . . 235  
 noquilt . . . . . 219, 227  
 norayexpmax . . . . . 238  
 norayfilter . . . . . 238  
 noraygauss . . . . . 238  
 norayjitter . . . . . 238  
 noraylog . . . . . 238  
 norendercombine . . . . . 227  
 normal . . . . . 217, 226  
 normalizeLine . . . . . 53, 255

normalizePlane . . . . . 60, 256  
 noscribemode . . . . . 241  
 noshadedbg . . . . . 208  
 nosmoothcrvs . . . . . 206, 242  
 notessgoodnorms . . . . . 230  
 notessholefill . . . . . 230  
 notessmaybequad . . . . . 230  
 notessnoref . . . . . 230  
 notessnormnorm . . . . . 230  
 notessquads . . . . . 230  
 notessquilt . . . . . 230  
 notransp . . . . . 217, 226  
 noviewport . . . . . 220, 227  
 novopacity . . . . . 218, 226  
 nowhite . . . . . 217, 226  
 nozkluge . . . . . 231  
 nth . . . . . 42, 252  
 nth coordinate menu item . . . . . 278  
 nTimes . . . . . 41, 252  
 number datatype . . . . . 24  
 number list menu item . . . . . 287  
 numbers menu . . . . . 275  
 numerical control machining . . . . . 166  
 numRamp . . . . . 41, 252  
 numStep . . . . . 41, 252

## O

object datatype . . . . . 24  
 object libraries . . . . . 150  
 object utilities menu . . . . . 275  
 objectsInDevWindow . . . . . 34, 251  
 objectsInWindow . . . . . 34, 251  
 ObjRefs!\* . . . . . 186  
 objTransform . . . . . 149, 267  
 of revolution menu item . . . . . 283  
 offset line menu item . . . . . 281  
 offlk . . . . . 220, 227  
 offset menu item . . . . . 279, 282, 284  
 offset plane menu item . . . . . 281  
 offset rbox menu item . . . . . 284  
 onlk . . . . . 220, 227  
 onelight . . . . . 217, 226  
 onePrinCircle . . . . . 81, 263  
 opacity . . . . . 303  
 opacity parameter . . . . . 223  
 open end condition . . . . . 11  
 open end condition menu item . . . . . 282, 283  
 open menu item . . . . . 287  
 open, chord, cubic menu item . . . . . 287  
 open, chord, quadratic menu item . . . . . 287  
 open, uniform, cubic menu item . . . . . 287  
 open, uniform, quadratic menu item . . . . . 287  
 optics attribute . . . . . 236  
 or . . . . . 312  
 order . . . . . 11

orientation conventions	29, 76
Origin	47
origin	246
origin menu item	279
Oslo Algorithm	12
other arc menu item	281
otherArc	65, 258
outside	29

## P

p2	61, 253
p2Pt datatype	25
p3	61, 253
p3Pt datatype	25
packedMatrix datatype	27
packedVector datatype	27
ParamByX	92
paramByX menu item	286
ParamByY	92
paramByY menu item	286
ParamByZ	92
paramByZ menu item	286
parametric information	69
parametric modeling	7, 19
parametric type	153
paramType datatype	26
parmEndCondType	71, 259
parmInfo	69, 259
parmInfo datatype	25
parminfo menu	275
parmKvType	71, 259
parmKvValue	71, 259
parmOrder	71, 259
partially bounded sets	6
periodic complete cubic menu item	284
periodic end conditions	12
periodic menu item	287
periodicCompleteCubicInterp	96, 267
perp line direction menu item	280
pick object type menu	275
plane datatype	25
plane normal menu item	280
plane pick menu item	277
planeNormal	60, 257
planeOffsetByDelta	59, 256
planes	58
planeThru2Lines	59
planeThruLineParallelToLine	59, 257
planeThruPtAndLine	59, 256
planeThruPtWithNormal	59, 257
planeThru2Lines	256
planeThru3Pts	59, 256
plastic surfaces	226
plus menu item	279
point & angle menu item	281

point & circle menu item	281
point & direction menu item	281
point & normal menu item	281
point datatype	25
point mesh menu item	279
point parallel menu item	281
point pick menu item	277
pointP	46, 253
points	44
points menu	275
polygon	57, 58, 256, 304
polygon datatype	25
polygon menu item	279
polygon pick menu item	278
polygonSize	256
polyline	57, 256, 304
polyline datatype	25
polyline menu item	279
polyline pick menu item	278
polylineFromPolygon	58, 256
polylineSize	256
polys	206, 242
predicate functions	46
prim datatype	26
primitives	85
primitives menu	275
prinCircles	81, 262
print object menu item	276
procedural modeling	19
procedures	312
profile	68, 260
profile & section menu item	283
profile curve	99, 102
profile menu item	282
profiled sweep menu item	283
programmer's interface	9
programming interface	20
project crv to plane menu item	282
project vec to plane menu item	280
projective by coords menu item	279
projective points	61
projectiveP	62, 253
projectPtOntoLine	55, 255
projPoint datatype	25
projPt	62, 252
prolog	242
PropagateChanges	39
prt ray tracing program	235
ps300 device	296
pseudocolor	217, 226
psheight	241
pt	44, 252
pt line distance menu item	278
pt plane distance menu item	278
pt project to line menu item	279

- pt project to plane menu item . . . . . 279  
 pt translate menu item . . . . . 285  
 ptBlend . . . . . 50, 254  
 ptDim . . . . . 46, 252  
 ptFrom3Planes . . . . . 60, 257  
 ptInterp . . . . . 49, 254  
 ptIntersectCircleLine . . . . . 67, 258  
 ptIntersectLineAndPlane . . . . . 60, 257  
 ptIntersect2Circles . . . . . 67, 258  
 ptIntersect2Lines . . . . . 55, 255  
 ptListFromCtlPoly . . . . . 75, 259  
 ptMinus . . . . . 49, 254  
 ptNearest2Lines . . . . . 55, 255  
 ptOffset . . . . . 48, 253  
 ptOnLine . . . . . 54, 255  
 ptOnLineNearestLine . . . . . 55, 255  
 ptOnLineWithX . . . . . 55, 256  
 ptOnLineWithY . . . . . 55, 256  
 ptOnLineWithZ . . . . . 55, 256  
 ptProjPtDirPlane . . . . . 60, 257  
 ptScaledOffset . . . . . 48, 254  
 ptSize . . . . . 46, 252  
 ptsOnLineList . . . . . 252  
 ptW . . . . . 61, 253  
 ptX . . . . . 44, 253  
 ptY . . . . . 44, 253  
 ptZ . . . . . 45, 253
- Q**
- QCrossSpread!\* . . . . . 105  
 qOffset . . . . . 139, 266  
 QRefMeasure!\* . . . . . 102  
 QUADRATIC . . . . . 71  
 quadratic menu item . . . . . 287  
 QUARTIC . . . . . 71  
 quartic menu item . . . . . 287  
 quick program . . . . . 234  
 quickaspect . . . . . 235  
 quickclear . . . . . 235  
 quickclip . . . . . 235  
 quickfb . . . . . 234  
 quicklexi . . . . . 234  
 quickoverlay . . . . . 235  
 quicktran . . . . . 235  
 quilt . . . . . 219, 227  
 QUINTIC . . . . . 71  
 quintic menu item . . . . . 287
- R**
- r2 . . . . . 45, 253  
 r2Vec datatype . . . . . 25  
 r3 . . . . . 46, 253  
 r3Vec datatype . . . . . 25  
 Radial . . . . . 92  
 radial menu item . . . . . 286  
 radialPattern . . . . . 165, 272  
 radius & 2 lines menu item . . . . . 281  
 radius circle line menu item . . . . . 281  
 radius menu item . . . . . 279  
 radius 2 circles menu item . . . . . 281, 282  
 radiusOfArc . . . . . 65, 258  
 raise degree menu item . . . . . 282  
 raiseCrvOrder . . . . . 73, 260  
 raiseCurve . . . . . 72, 260  
 raiseOrder . . . . . 72, 79, 260, 262  
 raiseSrfOrder . . . . . 79, 262  
 raiseSurface . . . . . 79, 262  
 ray tracing program . . . . . 235, 237  
 raybuff . . . . . 237  
 rayexpmax . . . . . 238  
 rayfilter . . . . . 238  
 raygauss . . . . . 238  
 raygrid . . . . . 238  
 rayheight . . . . . 239  
 rayjitter . . . . . 238  
 raylog . . . . . 238  
 raysample . . . . . 238  
 raysize . . . . . 239  
 raythresh . . . . . 238  
 raywidth . . . . . 239  
 RBedge . . . . . 90  
 rBox datatype . . . . . 26  
 rbox keywords menu . . . . . 275  
 rboxFrom6Planes . . . . . 89, 263  
 rboxOffsetFromRbox . . . . . 91, 263  
 rCone datatype . . . . . 26  
 rCylinder datatype . . . . . 26  
 RDedge . . . . . 90  
 rectangularPocket . . . . . 165, 272  
 refine menu item . . . . . 282, 283  
 refinement . . . . . 12, 72, 76  
 reflect . . . . . 43, 260, 264  
 reflect menu item . . . . . 282  
 reflect point menu item . . . . . 279  
 reflections . . . . . 43  
 region warp menu item . . . . . 285  
 regional warp . . . . . 131  
 regionFromCrv . . . . . 75, 261  
 regionFromSrf . . . . . 79, 262  
 regionWarp . . . . . 131, 264  
 regionWarpR . . . . . 131, 264  
 remAttr . . . . . 181, 270  
 remove window menu item . . . . . 277  
 removeDevWindow . . . . . 34, 251  
 removeWindow . . . . . 33, 251  
 render . . . . . 302  
 render program . . . . . 215, 216  
 rendercombine . . . . . 227  
 repeat statement . . . . . 313  
 replace instance menu item . . . . . 285

replace object menu item	276	rotateY	143, 267
replace point menu item	276	rotateYAxis	145, 268
replaceDescriptor	148, 149, 270	rotateYAxisWithX	145, 269
replaceObject	149, 267	rotateYAxisWithZ	146, 269
reShow	35, 249	rotateZ	143, 268
reshow list menu item	276	rotateZAxis	145, 268
reshow object menu item	276	rotateZAxisWithX	146, 269
resolution	303	rotateZAxisWithY	146, 269
resolution parameter	222	rotational sweeping	83
restore	186, 271	rotations	143
restore model menu item	278	rounded box menu item	283
restriction datatype	27	rounded cylinder menu item	283
restriction lists	133	rounded edge box	88
return statement	313	rounded truncated cone menu item	283
reverse menu item	281, 282, 283	roundRightCirCylinder	88, 263
reverse normals menu item	277	roundTruncCirCone	88, 263
reverseArc	257	ROW	75
reverseCrv	261	row menu item	286
reverseKv	259	rowParmInfo	77
reverseLine	255	rowParmInfo	261
reverseMeshInDir	259	RUedge	90
reverseObj	43, 76, 271	ruled menu item	283
reversePlane	257	ruled surfaces	98
reversePolygon	256	ruledSrf	98, 265
reversePolyline	256	run-length encoded files	219
reverseSrf	262	run-length-encoded files	205
reverseSrfInDir	76, 261	rx	143, 269
reversing orientation	43	ry	143, 269
revolution	83	rz	143, 269
RFACE	89		
RFedge	90	<b>S</b>	
right face menu item	286	s3	144, 270
right menu item	286	sameOrder	73, 261
right mouse button in GUI	22	save model in window menu item	278
rightAngleWedge	86, 263	save model menu item	278
rightCirCylinder	86, 263	save/restore menu	275
RLE file	205	saveModel	186, 271
Rlisp syntax	311	saveModelInWindow	186, 271
rlisp-pgm	315	scalar statement	313
rnVec datatype	25	scale menu item	280, 285
rotate menu item	285	scaled offset menu item	279
rotateLine	143, 268	scaleP3	268
rotateVectorToXAxis	145, 268	scaleUniform	144, 268
rotateVectorToXAxisWithY	146, 269	scaleX	144, 268
rotateVectorToXAxisWithZ	146, 269	scaleXYZ	144, 268
rotateVectorToYAxis	145, 268	scaleY	144, 268
rotateVectorToYAxisWithX	146, 269	scaleZ	144, 268
rotateVectorToYAxisWithZ	146, 269	screen utilities menu	275
rotateVectorToZAxis	145, 268	screen_trans	300
rotateVectorToZAxisWithX	146, 269	screens	31
rotateVectorToZAxisWithY	147, 269	second	42, 252
rotateX	143, 267	segBetween2Lines	55, 255
rotateXAxis	145, 268	select window menu item	277
rotateXAxisWithY	145, 268	selectDevWindow	33, 251
rotateXAxisWithZ	145, 268	selected window set	32

- selectWindow . . . . . 33, 251
- seq0 . . . . . 42, 252
- seq1 . . . . . 42, 252
- set expression . . . . . 191, 196
- set operations . . . . . 5
- set rbox faces open menu item . . . . . 284
- set rbox radius menu item . . . . . 283
- set\_expr . . . . . 309
- setAttr . . . . . 181, 270
- setCrvFineness . . . . . 37, 250
- setCrvPolys . . . . . 37, 249
- setdatadir . . . . . 190
- setDevWindowViewMat . . . . . 39, 251
- setIsoLines . . . . . 37, 250
- setRboxFacesOpen . . . . . 89, 263
- setRboxRadius . . . . . 90, 263
- setReverseNorms . . . . . 38, 250
- setSmoothCrvs . . . . . 37, 249
- setSrfFineness . . . . . 38, 250
- setSrfMeshes . . . . . 37, 250
- setSrfNorms . . . . . 38, 250
- setWindowViewmat . . . . . 39
- setWindowViewMat . . . . . 251
- setxres . . . . . 220, 227
- setyres . . . . . 220, 227
- sg . . . . . 144, 269
- shade program . . . . . 205, 207
- shadedbg . . . . . 208
- shape operations menu . . . . . 275
- shape operators . . . . . 122
- shape\_edit program . . . . . 9
- shell . . . . . 181, 191, 195, 196, 267, 308
- shell datatype . . . . . 26
- shell menu item . . . . . 285
- shellAxisProfileSection . . . . . 84, 265
- shellOfRevolution . . . . . 84, 265
- show . . . . . 35, 249
- show object menu item . . . . . 276
- show prereqs menu item . . . . . 276
- show restored model menu item . . . . . 278
- show\_defaults . . . . . 215
- show\_hiddefaults . . . . . 232
- ShowAll . . . . . 36
- ShowAssignments . . . . . 36
- showHigh . . . . . 36, 249
- showInWindow . . . . . 35, 249
- showNamedModel . . . . . 186, 271
- showPrereqs . . . . . 36, 249
- showRestoredModel . . . . . 186, 271
- shrinkIt . . . . . 38, 250
- signedDistPtLine . . . . . 56, 255
- signedDistPtPlane . . . . . 60, 257
- singleSrfEdge . . . . . 115, 264
- skeletal warp . . . . . 129
- skeletal warp menu item . . . . . 285
- skelWarp . . . . . 129, 265
- skelWarpR . . . . . 131, 265
- sKvs . . . . . 78, 261
- sKvTypes . . . . . 78, 261
- slots in objects . . . . . 153
- slottedHole . . . . . 165, 272
- sMesh . . . . . 78, 261
- smooth . . . . . 217, 226
- smooth curves menu item . . . . . 277
- SmoothCrvs . . . . . 36
- smoothcrvs . . . . . 206, 242
- solid modeling . . . . . 3
- sOrders . . . . . 78, 261
- sParmInfos . . . . . 77, 261
- sphere . . . . . 87, 263
- sphere datatype . . . . . 26
- sphere menu item . . . . . 283
- spline functions menu . . . . . 276
- square\_aspect . . . . . 220, 227
- sRefine . . . . . 76, 262
- srf\_mash . . . . . 229
- srfAndDerivs . . . . . 82, 262
- srfAxisProfileSection . . . . . 83, 265
- srfData datatype . . . . . 27
- srfDivide . . . . . 80, 262
- srfEdge . . . . . 114, 265
- srfEval . . . . . 80, 262
- SrfFineness . . . . . 37
- srfFineness . . . . . 206, 242
- srfFromCrvs . . . . . 80, 261
- srfFromCrvsDir . . . . . 80, 261
- srfGeom . . . . . 81, 262
- srfMerge . . . . . 80, 262
- SrfMeshes . . . . . 37
- srfOfRevolution . . . . . 83, 265
- srfSubdiv . . . . . 81, 262
- startDirOfArc . . . . . 65, 258
- state . . . . . 215
- step statement . . . . . 313
- stretch . . . . . 123, 265
- stretch menu item . . . . . 284
- string datatype . . . . . 24
- string pick menu item . . . . . 278
- string\_attr . . . . . 303
- stringAttr . . . . . 182, 270
- sTypes . . . . . 78, 261
- subdivision . . . . . 15
- subtract vectors menu item . . . . . 280
- subWorld . . . . . 235, 271
- subworlds . . . . . 235
- surface . . . . . 75, 261
- surface and polygon normals . . . . . 29
- surface boundary menu item . . . . . 282
- surface datatype . . . . . 25
- surface fineness menu item . . . . . 277

surface menu item	282
surface meshes menu item	277
surface normals menu item	277
surface orientation	76
surface pick menu item	278
surface refinement	76
surface region menu item	283
surface_quality	302
surface_texture	302
surfaceNormals	81, 262
surfaceOpen	76, 262
surfaceQuality	236, 271
sweep menu item	283
sweepConstantWidth	102, 265
sweeps	99
sweepWithProfile	103, 265
switchTo	32, 250
sx	144, 270
sxyz	144, 269
sy	144, 270
symbol datatype	24
sz	144, 270

## T

table datatype	27
tangent lines menu item	282
taper	123, 265
taper menu item	284
taperWithSplineFn	126, 266
colors program	204
tensor-product	15
tess	230
tessgoodnorms	230
tessholefill	230
tessmaybequad	230
tessnoref	230
tessnormnorm	230
tessquads	230
tessquilt	230
tessres	230
text	39, 271
text file format	299
text_justify attribute	243
text_string	309
textObj datatype	27
texture attribute	236
then statement	313
thicken menu item	284
third	42, 252
times menu item	279
top menu item	286
torus	87, 263
torus datatype	26
torus menu item	283
transform	143, 147, 267

transform datatype	26
transform object menu item	286
transformation trees	147
transformations menu	275
transforming objects	143
translate menu item	285
translate xyz menu item	285
translateX	144, 268
translateXYZ	143, 144, 268
translateY	144, 268
translateZ	144, 268
transp	218, 226
transparency parameter	223
truncated cone menu item	283
truncRightCone	86, 263
tube menu item	283
twist	125, 266
twist menu item	284
twistWithSplineFn	126, 266
twolight	217, 226
twoperflat	219, 227
tx	143, 269
ty	143, 269
tz	143, 269

## U

UBedge	90
UFACE	89
UFedge	90
unblended sweeps	99
unHighLight	36, 249
unhighlight object menu item	276
Uniform	92
uniform menu item	286, 287
uniform scale menu item	285
unit circle menu item	281
unit sphere menu item	283
unit vector menu item	280
unitCircle	67
unitVec	48, 253
unShow	35, 249
unshow list menu item	276
unshow object menu item	276
unshow prereqs menu item	276
unShowFromWindow	249
unShowInWindow	35
unShowPrereqs	36, 249
until statement	313
up face menu item	286
user attributes	182
using parminfo menu item	284
utility programs	9

## V

vec	44, 252
-----	---------

vecAngle . . . . . 47, 253  
 vecAtAngle . . . . . 47, 253  
 vecBlend . . . . . 50, 254  
 vecDim . . . . . 46, 252  
 vecFrom2Pts . . . . . 254  
 vecInterp . . . . . 49, 254  
 vecLength . . . . . 47, 253  
 vecMinus . . . . . 48, 253  
 vecOffset . . . . . 49, 254  
 vecOrigin . . . . . 47, 252  
 vecP . . . . . 46, 253  
 vecPlus . . . . . 48, 253  
 vecProjVecOntoPlane . . . . . 61, 257  
 vecScale . . . . . 49, 254  
 vecSize . . . . . 46, 253  
 vector length menu item . . . . . 278  
 vector pick menu item . . . . . 277  
 vector to axis menu item . . . . . 285  
 vector to axis orient menu item . . . . . 285  
 vectorOf datatype . . . . . 24  
 vectors . . . . . 44  
 vectors menu . . . . . 275  
 vecX . . . . . 45, 253  
 vecY . . . . . 45, 253  
 vecZ . . . . . 45, 253  
 vertex . . . . . 303  
 vertical menu item . . . . . 280  
 view program . . . . . 203, 206  
 view\_distance attribute . . . . . 236  
 view\_size attribute . . . . . 236  
 view\_trans . . . . . 300  
 viewiris . . . . . 206  
 viewMat . . . . . 38, 251  
 viewMat datatype . . . . . 26  
 viewport . . . . . 220, 227  
 viewps . . . . . 206  
 viewX . . . . . 207  
 view300 . . . . . 206  
 view300mat . . . . . 207  
 vOffset . . . . . 113, 265  
 volume representation . . . . . 3  
 vopacity . . . . . 218, 226

## W

W coordinate menu item . . . . . 278  
 warp . . . . . 127, 265

warp menu item . . . . . 285  
 warpR . . . . . 129, 265  
 wedge . . . . . 86  
 wedge datatype . . . . . 26  
 wedge menu item . . . . . 283  
 while statement . . . . . 313  
 white . . . . . 217, 226  
 width . . . . . 303  
 width parameter . . . . . 222  
 window manager . . . . . 34  
 windows . . . . . 31, 32

## X

x axis menu item . . . . . 280  
 X coordinate menu item . . . . . 278  
 X direction menu item . . . . . 280  
 x menu item . . . . . 286  
 XAxis . . . . . 56  
 Xdefaults file . . . . . 293  
 XDir . . . . . 47  
 xgen device . . . . . 292  
 Xray program . . . . . 240  
 xraymag . . . . . 240  
 xy plane menu item . . . . . 281  
 xyz scale menu item . . . . . 285  
 xz plane menu item . . . . . 281

## Y

y axis menu item . . . . . 280  
 Y coordinate menu item . . . . . 278  
 Y direction menu item . . . . . 280  
 y menu item . . . . . 286  
 YAxis . . . . . 56  
 YDir . . . . . 47  
 yes menu item . . . . . 286  
 yz plane menu item . . . . . 281

## Z

Z coordinate menu item . . . . . 278  
 Z direction menu item . . . . . 280  
 z menu item . . . . . 286  
 ZAxis . . . . . 56  
 ZDir . . . . . 47  
 zkluge . . . . . 231

## 1. Tutorial

This tutorial is designed to give a new user of Alpha\_1 an overview of how one normally interacts with the system. Enough information is given so that the reader can try out every operation described, but the explanations are limited so that the overall direction is not lost in the detail. The tutorial is not a synopsis of the User's Manual; to get a full idea of what operations are available it is necessary to read the User's Manual or skim over the appendix that summarizes the routines.

After this tutorial, a user should be able to interact with the command-driven interface to `shape_edit` from within emacs, create simple models and display them either as line drawings or shaded raster images, use simple boolean operations, and save the results of his work for use at a later time.

### 1.1 Preliminaries

This tutorial assumes that you have at least a minimal knowledge of Unix and emacs. If you don't know how to log in, find some files, and edit them, you'll probably be lost from the beginning.

The tutorial also assumes that you have a running Alpha\_1 system on which to try these examples. No pictures of construction results are included, since your graphics display will generate the illustrations as you work through the tutorial.

Before you are able to try the operations described in this tutorial you will have to have some setup files. The easiest way to get these is to have an Alpha\_1 systems person run the `setup-new-user` program for you. If this is not possible or desirable, the following is the setup you need to do:

1. Make sure you can access the Alpha\_1 programs. If you're not sure, type `shape_edit` to the unix shell and see if you get the `shape_edit` prompt. To exit the `shape_edit` (should you succeed), type `exitlisp()`; to `shape_edit`.
2. Copy the following files from "`$alpha1/cmds`" to your home directory:

`.login .cshrc .aliases .defs .aldefs .logout .rlisprc .emacs_pro .emacs`

Note: If you already have your own versions of these files, then you may want to merge in the Alpha\_1 versions so as not to lose your own login commands. If you will be using an X window manager, you may also want to copy "`.Xdefaults`" and "`.uwmrc`" from the same directory.

3. Edit the "`.login`" file and change "`USERNAME`" to your name.

### 1.2 Using Shape\_edit Under Emacs

Construction of models in the Alpha\_1 system is done with an interactive program called `shape_edit`. Normally, we run `shape_edit` as a process under an emacs editor. This allows the commands given to `shape_edit` to be revised and re-executed without retyping them. It also allows a script of commands for generating a model to be accumulated as the model is created.

The easiest way to start the `shape_edit` from your emacs is to read in a file with a ".r" extension. Try finding a file called "`test.r`" and notice that when emacs creates the buffer it puts you in "`rlisp`" mode. Now type

```
2+2;
```

into that buffer. Leave the cursor anywhere on the line and execute the emacs command "`M-e`" (meta-e or escape-e). This executes the `rlisp` line which your cursor is on. If you don't already



have a `shape_edit` process running, it will start one for you, popping up another window to show the results. In this case the result is just 4.

You can now edit the line to change the expression, or add more lines to the buffer and execute those. Remember that you must say "M-e" to get the line executed — carriage return just puts a carriage return in the buffer, but doesn't execute the line. Also note that the semicolon is crucial. It denotes the end of the statement, and `shape_edit` won't try to evaluate what you have said unless it finds a semicolon. In fact, it will continue to read lines out of your buffer until it finds a semicolon or runs out of lines.

Comments in `shape_edit` are denoted by the "%" character. Comments will be used throughout the rest of this tutorial to explain `shape_edit` commands. Obviously, you don't need to type them!

Some slightly more interesting things to try in `shape_edit` are:

```
P1 := pt( 1, 2 );      % Create a point, and call it P1.
P2 := pt( 2, 2 );
LineA := lineHorizontal( 2.0 ); % Horizontal line at y=2.
LineB := lineThru2Pts( P1, P2 ); % Line through P1 and P2.
```

### 1.3 Displaying Geometry

As soon as you begin constructing graphical objects, you will want to see them on the graphics screen instead of looking at numbers in the `shape_edit` window. First you have to tell `shape_edit` what graphics device you want to use. The devices supported by `shape_edit` are described in an appendix of the User's Manual. Supposing you have a PS300, you would say

```
grab ps300;
```

to tell `shape_edit` that you wanted to use that device.

Most devices have a large set of hardware dependent controls for managing windows and controlling viewing. The only things you really need to know in order to get started on a device are (1) how to create a window and (2) how to scale the viewing. On devices that don't really have a window manager yet, the window usually just pops up on the screen in a default size and position. For devices with window managers, you will probably get a prompt which indicates that you should use the tablet or mouse to designate the upper left and lower right corners of the window. Knowing how to scale the view is important because the default display area is -1.0 to 1.0 in X and Y, but most objects extend beyond that area when the most convenient coordinate system is used. On devices with knobs, one of the knobs will be tied to the scaling. Other devices may have slider bars on the display or function keys used in conjunction with a mouse or tablet.

In order to get objects displayed on the device once it has been grabbed, we need to "show" them. This can be done explicitly with the `show` command or implicitly when the object is created using a special form of assignment statement. Try the following commands (typing them into your "test.r" buffer, and executing "M-e" for each line, as before):

```
show origin;          % Point at (0,0) is named origin.
show( P1, P2 );       % We created these earlier.
P3 := pt( 1, 1 );
Crv1 := profile( P1, P2, P3, P1 ); % Make a curve connecting the points.
```

Note the different syntax for `show`. `Shape_edit` allows you to leave off the parentheses in a function call if there is exactly one argument. So the first statement above is exactly equivalent to

```
show( origin );
```

If you have more than one argument, you will always have to include the parentheses.

The creation of P3 shows the use of the special “^=” assignment statement. It is just like the “:=” we used above, except that as a side effect it displays the object on the graphics device. Note that since P1 and P2 have Y coordinates equal to 2, you will have to scale down the view so that you can see them.

In the next section, we will begin an example of the construction of a non-trivial model: a spoon. To prepare for that, let’s clear the screen of what we have drawn so we can start fresh. We could “unshow” all the objects we displayed using `unshow`, but an easier way is to use `clearWindow`.

```
unshow( P1, P2 );
clearWindow Default;
```

“Default” is the name of the window which is automatically created for you if you didn’t specifically create one yourself. We could have used `newWindow` at the beginning to create a window called “Test”.

```
newWindow Test;
```

## 1.4 Constructing a Model

The modeling example we use in this section comes from the `shape_edit` examples directory: “\$she/spoon.r”. Rather than typing in all the `shape_edit` commands shown here, you should read “spoon-tutorial.r” into your emacs buffer, and execute the commands as you read along.

### 1.4.1 Flat Bowl Construction

The spoon is constructed in two parts: the bowl shape, then the handle. We first construct the outline of the bowl of the spoon. The commands below use a set of lines to “frame” the shape of the spoon silhouette, and then form an initial curve from the points at the intersections of the lines.

```
HandleLine ^= lineHorizontal(-0.25 );
BowlLine ^= lineHorizontal(-1.0 );
Endline1 ^= lineVertical( 0.0 );
EndLine2 ^= lineVertical( 3.0 );

% These points will be the control points for a cubic curve.
Pt1 ^= ptIntersect2Lines( HandleLine, Endline1 );
Pt2 ^= ptIntersect2Lines( HandleLine, lineVertical( 0.5 ) );
Pt3 ^= ptIntersect2Lines( BowlLine, lineVertical( 0.75 ) );
Pt4 ^= ptIntersect2Lines( BowlLine, lineVertical( 2.7 ) );
Pt5 ^= ptIntersect2Lines( EndLine2, HandleLine );

% Form the first curve.
BottomCrv ^= curve( parminfo( cubic, ec_open, kv_uniform ),
                    list( Pt1, Pt2, Pt3, Pt4, Pt5 ) );
```

The top curve is merely a reflection of the first curve, made by negating the Y coordinates:

```
TopCrv ^= reflect( BottomCrv, 'x );
```

In order to create a closed boundary, we need to construct some curves on the left and right which meet the endpoints of the top and bottom curves. The one on the left should just be a straight line, and an easy way to build a straight line curve is using `profile`.

```
LeftCrv ~= profile( crvPt( TopCrv, 0 ), crvPt( BottomCrv, 0 ) );
```

The curve on the right should be tangent to the top and bottom curves, so we build two intermediate points which will be used to form a cubic curve:

```
% Remember the size of the control polygon of the bottom curve.
SideSize := cPolySize BottomCrv->cPoly - 1;
```

```
% The right end curve must come in tangent to the sides.
EndPt1 ~= ptInterp( crvPt( TopCrv, SideSize-1 ),
                    crvPt( TopCrv, SideSize ),
                    1.15 );
EndPt2 ~= ptInterp( crvPt( BottomCrv, SideSize-1 ),
                    crvPt( BottomCrv, SideSize ),
                    1.15 );
```

To construct a spline curve from a set of points which form the control polygon, we must specify the parametric information for the curve and the ordered set of points. In this case, the resulting curve will be cubic, with open end conditions and a uniform knot vector. These are usually reasonable default choices for the parametric information. The easiest way to create a set of points is to make them into a list as is done below:

```
RightCrv ~= curve( parminfo( cubic, ec_open, kv_uniform ),
                  list( crvPt( TopCrv, SideSize ),
                        EndPt1, EndPt2,
                        crvPt( BottomCrv, SideSize ) ) );
```

Now we would like to make a surface from these four boundary curves. One way to do it is with the `boolSum` (for boolean sum) operation:

```
FlatBowl ~= boolsum( TopCrv, BottomCrv, LeftCrv, RightCrv )$
FlatBowl := ptObjCoerce( FlatBowl, 'e3pt )$ % Make the surface 3D.
```

If we hadn't wanted the right end to be rounded, we could have used `ruledSrf`. You might want to try `ruledSrf` and compare the two surfaces.

```
Tmp ~= ruledSrf( TopCrv, BottomCrv );
```

At this point, you might want to clear away the construction on the screen, leaving only the flat bowl surface. (Or you could have created a new window in which to continue the rest of the spoon construction.)

```
clearwindow default;
show FlatBowl;
```

## 1.4.2 Shaping the Bowl

If you use the viewing controls on your display, you will see that what we have constructed so far looks nice from the top, but still lies in a plane, and so doesn't make a very good spoon. The `warp` operation will let us shape the bowl, sort of like pushing on clay with a finger. The lines below set up some parameters for the warping, and then "refine" the flat bowl surface. Refining a surface just adds extra control points without changing the shape, so when you want to change the shape you have more control.

```

WarpDirection := r3Vec( 0.0, 0.0, -0.50 ); % Push towards -Z.
WFac := 2.0; % Warp factor controls shape of bump.

```

```

FlatBowl := sRefine( FlatBowl, ROW, '( 0.55 ), T )$
FlatBowl := sRefine( FlatBowl, COL, '( 0.5 ), T )$

```

```

CtrPt ~= FlatBowl->sMesh[2][3]; % Center of warp.

```

When we do the warp, we only want to affect control points that don't lie on the boundary of the bowl, because we don't want to change the silhouette shape that be so carefully constructed. So we will first build a "lasso" to surround the interior points, and tell the warp operator to restrict itself to points in the region. It will be easier to construct the region if we look at just the control points rather than the isoparametric lines.

```

setIsolines( nil );
setSrfMeshes( t );
reshow FlatBowl;

```

```

P1 ~= ptOffset( FlatBowl->sMesh[1][4], vec( 0.03, 0.08 ) );
P2 ~= ptOffset( FlatBowl->smesh[1][3], vec( 0.0, 0.08 ) );
P3 ~= ptOffset( FlatBowl->smesh[1][2], vec( -0.03, 0.08 ) );
P4 ~= ptOffset( FlatBowl->smesh[3][2], vec( -0.03, -0.08 ) );
P5 ~= ptOffset( FlatBowl->smesh[3][3], vec( -0.0, -0.08 ) );
P6 ~= ptOffset( FlatBowl->smesh[3][4], vec( 0.03, -0.08 ) );

```

```

Region ~= polyline( list( P1, P2, P3, P4, P5, P6, P1 ) )$

```

```

WarpBowl ~= warpR( FlatBowl, WarpDirection, CtrPt, 1.0, 1.8, nil,
    buildRestriction( list list( 'inside, region, 'xy ) )
)$

```

```

setIsolines( t ); % Go back to isoparametric lines.
setSrfMeshes( nil );
reshow WarpBowl;

```

You may want to again clear away everything but the most recent construction.

```

clearWindow default;
show WarpBowl;

```

We have now completed the basic shape of the bowl, and are almost ready to add the handle. First though, we want to shape the end of the bowl where the handle will be attached. The bowl should slope up at that point, rather than lying in the plane. We can do this with a bend operation.

```

BendAngle := 34.4; % Angle in degrees.
FinalBowl ~= bend( WarpBowl, 'x, BendAngle, 0.5, -0.1, 0.5, T )$

```

Let's also put a little warp in the end so that the handle doesn't have to be completely flat.

```

LittleWarp := r3vec( 0.0, 0.0, 0.10 );
FinalBowl ~= warp( FinalBowl, LittleWarp,
    FinalBowl->sMesh[2][0], 2.0, 0.25, nil )$

```

### 1.4.3 Adding the Handle

To make the handle, we will first extract the curve at the end of the bowl so we can make the two pieces join up. Using a tangent from the surface, we'll just extrude that curve to make an initial handle surface.

```
HandleDir := vecFrom2Pts( FinalBowl->sMesh[2][0],
  FinalBowl->sMesh[2][1] );
EndCrv ~= crvFromSrf( FinalBowl, COL, 0 );

FlatHandle ~= reverseObj extrudeDir( EndCrv,
  vecScale( HandleDir, -10.0 ) )$

% Turn it end-for-end, and raise the degree to cubic.
FlatHandle ~= reverseSrfInDir( FlatHandle, row )$
FlatHandle ~= raiseSurface( FlatHandle, ROW, cubic )$
```

We now have two surfaces, which match at the boundary. In order to make the slopes continuous across the join and to make it easier to work with, we will merge the surfaces into a single spline surface.

```
SpoonSrf ~= srfMerge( FlatHandle, FinalBowl, ROW, 'EXACT )$
SpoonSrf->sParminfos[0] := parminfo( cubic, ec_open, kv_uniform );
reshow SpoonSrf;
```

The handle clearly needs some shaping. We use the taper function to slim down the handle in the center. The range of the handle runs from -4.3 to 0.5 in X, so we want to taper handle in in that range. We have to define a little function that will be called to determine the scaling values for each control point along the X axis.

```
procedure HandleTaper( Val, ScaleVal );
  if ( Val > 0.4 ) then 1.0
  else if ( Val > -1.8 ) then ScaleVal
  else 1.0;

SpoonSrf2 ~= taper( SpoonSrf, 'x, NIL, 'HandleTaper, list( 0.5 ) )$
clearWindow default;
show SpoonSrf2;
```

The last argument to **taper** is a list of arguments which the **handleTaper** procedure will use in addition to the X coordinate of each control point to determine how to scale the Y coordinates. In this example, control points on the surface which lie between -1.8 and 0.4 in X will have their Y coordinates halved. You may wish to try the taper with other values than 0.5 for the scaling, or with other range limits to explore the effects on the handle.

Finally, if you look at the spoon from the side, you will see that we need to bend the handle down so it looks more like a proper spoon and less like a soup ladle. We must first add some extra control points so that the bend will have enough points to make a nice result.

```
HandleRefKv := '( 0.5 1.5 2.5 3.5 );
SpoonSrf2 := sRefine( SpoonSrf2, ROW, HandleRefKv, T )$

BendAngle2 := -20.0;
FinalSpoon ~= bend( SpoonSrf2, 'x, BendAngle2, 0.2, -0.8, 0.2, T )$
unshow SpoonSrf2;
```

## 1.5 Making a Raster Picture

We will consider our spoon model complete at this point, even though it actually needs a little more work to make a true model that encloses some space. What we have designed so far is just a surface: it has no thickness. However, we want to cover some other aspects of Alpha\_1 in the remainder of this tutorial.

One of the most important utilities you will need to know how to use is the **render** program for producing shaded raster images of your models. Smoothly shaded color pictures can often give much more information about the shape of an object than line drawings can. In order to run the rendering program, we must first save our model data to a file from **shape\_edit**. We always use the ".a1" file extension to identify the text data which comes from **shape\_edit**. If you had more than one surface you wanted to put in the file you would have to make a list of them to pass to **dumpA1File**:

```
dumpA1File( FinalSpoon, "spoon.a1" );  
dumpA1File( list( FinalSpoon, OtherSrf ), "spoon.a1" );
```

Now you are ready to leave **shape\_edit** and get ready to run the **render** program from the Unix shell. The most important thing you have to tell **render**, aside from the data itself, is the view in which you want to see the data. Our spoon, for example, will be much more interesting if we view it from an oblique angle instead of directly from the top. A **view** program is available for selecting and storing viewing parameters. You will first need to load the viewing commands with **getcmds** and then choose a display device. We might say **view300** to run the program on the PS300. The default is to run under an X window manager. Run the **view** program on your data by just giving **view** the name of the data file.

```
getcmds view  
view300  
view spoon.a1
```

If you were using the same device in your **shape\_edit** session as the one you are trying to run **view** on, you may get a message saying the device isn't available or doesn't exist. This is because your **shape\_edit** session is still running, and some devices can only be used by one process at a time. You will need to drop the device from **shape\_edit** using the **drop** command to release it for other processes to use. When you need it again from **shape\_edit**, you can use **grab**, just like you did at the beginning, and all your data will be restored on the device.

```
drop ps300;
```

Once **view** is running, you can twist the knobs and experiment with different views interactively until you find one you like. Then you need to save the viewing transformation. On the PS300, run **get300mat** and press the "writemat" function button, followed by the "quit" function button. (This puts the output in a file called "a.mat" which you will probably want to copy to a different name.) On most other devices, buttons to write the matrix are included with **view**.

Now you are ready to run the rendering program. You will first need to load the rendering commands using **getcmds** again:

```
getcmds render
```

This only needs to be done once in a session. The commands will continue to be available until you log out. We use the ".rle" extension to denote image files created by **render** (because they are stored in a run-length encoded format). Start the rendering process by giving **render** the input files and redirecting the standard output to a ".rle" file. Since **render** usually takes a little while, the "&" on the end of the line tells the shell to do it as a background process so that you can do

other things while it is computing. In particular, you can check how **render** itself is progressing using **getfb** to get the image on the frame buffer. (If you have a different device, the image display program may have a different name: **getiris**, **getbob**, **getx** are some of them.)

```
render spoon.mat,spoon.ai > spoon.rle &
getfb spoon.rle
```

## 1.6 Simple Boolean Combinations

One other important utility of Alpha\_1 which you may need to use is the **combine** program for calculating boolean combinations (union, intersection, and difference) of parts of your model. We will need to back up into **shape\_edit** again and create a few more pieces of geometry to demonstrate the boolean operations. Let's use them to create a slotted spoon out of the spoon model we already created.

What we want to design to create slots is a surface which will cut out the slot shape when we combine it with the spoon surface. You can think of it as a cookie cutter for this example. We begin by viewing the spoon from the top again, and designing a simple curve from a set of points we create:

```
SlotCrv ^= reverseObj curve( parminfo( cubic, ec_open, kv_uniform ),
    list( pt( 1.0, 0.0 ),
          pt( 1.0, 0.08 ),
          pt( 1.3, 0.08 ),
          pt( 2.5, 0.08 ),
          pt( 2.8, 0.08 ),
          pt( 2.8, 0.0 ) ) )$
SlotCrv2 ^= crvConcat( SlotCrv, reverseObj reflect( SlotCrv, 'y' ) )$

SlotSrf ^= extrudeDir( SlotCrv2, vec( 0, 0, -1.0 ) )$
```

We would like three slots, but the other two are just instances of the first one.

```
SlotSrf1 ^= objTransform( SlotSrf, ty( 0.001 ) )$
SlotSrf2 ^= objTransform( SlotSrf, sg( 0.8 ), tx( 0.375 ), ty( -0.3 ) )$
SlotSrf3 ^= objTransform( SlotSrf, sg( 0.8 ), tx( 0.375 ), ty( 0.3 ) )$
```

In order for the combiner to know that we intended the surfaces to join back to themselves, we must declare the two boundaries of each slot to be adjacent using **mkAdjacent**. Adjacencies must always be declared across boundaries that join up this way.

```
mkAdjacent( SlotSrf1, 'left', SlotSrf1, 'right' );
mkAdjacent( SlotSrf2, 'left', SlotSrf2, 'right' );
mkAdjacent( SlotSrf3, 'left', SlotSrf3, 'right' );
```

Now, all the geometry is ready for the slotted spoon, and we only need to tell the combiner how we want it used. The combiner works with "shells" which are groups of surfaces which interact to form the resulting part. In our case, the three slots don't interact with each other, so they will form one shell, and the spoon surface will form the other shell.

```
SpoonShell := shell( list( FinalSpoon ) )$
SlotShell := shell( list( SlotSrf1, SlotSrf2, SlotSrf3 ) )$
```

Finally, we create an expression that describes how we want these objects combined, and dump the expression to a file.

```
SpoonExpr := combinerObject( SpoonShell-SlotShell )$
dumpA1File( SpoonExpr, "slot-spoon.a1" );
```

At the Unix shell, we can invoke the combiner on this data to actually evaluate the results. (The default combiner epsilon, for determining how closely to evaluate the results, is too coarse for most models, so we set it to a finer value.)

```
getcmds combine
comb_eps 0.01
combine slot-spoon.a1 "" >slot-spoon.polys -c "spoonshell"
```

The string value can be used to specify the combiner expression, but since we already put it in the file, we leave it empty. The "-c" flag says to output only the `SpoonShell` surface, not any parts of the surfaces we used for the slots. The output is sent to a file with the ".polys" extension to indicate that it came from the combiner program. To check that you got the desired result, you can view the "slot-spoon.polys" file:

```
view slot-spoon.polys
```

You could also make a shaded rendering of the slotted spoon. We can use the same viewing parameters (in "spoon.mat") if we want because the basic spoon data is the same.

```
render spoon.mat,slot-spoon.polys > slot-spoon.rle &
getfb slot-spoon.rle
```

## 1.7 Finishing Up

Now that you have completed your model, you will need to know what things are important to keep.

The most important file is the ".r" file that you created in your emacs buffer, since this contains all the `shape_edit` commands you used to generate the model. This file can be loaded back into your emacs and the commands re-executed (and modified) whenever you like.

We often keep the ".a1" files as well, because they are sort of an evaluated and cached form of the ".r" file.

In general, images (".rle" files) are saved only as long as they are needed — they take up lots of room on the disk.

```
rm spoon.rle slot-spoon.rle
```

The same is true for the combiner output (".polys" files). The combiner output is generally simple to regenerate if necessary, especially if you have the combiner expression stored with the data as we did in our spoon example.

```
rm slot-spoon.polys
```

This is the end of the tutorial.





# **The Utah Raster Toolkit**

## ***Contents:***

- *The Utah Raster Toolkit*
- *Design of the Utah RLE Format*
- Toolkit User's Guide
- Toolkit Programmer's Guide





# The Utah Raster Toolkit

John W. Peterson  
Rod G. Bogart  
*and*  
Spencer W. Thomas

University of Utah, Department of Computer Science<sup>1</sup>  
Salt Lake City, Utah

## Abstract

The Utah Raster Toolkit is a set of programs for manipulating and composing raster images. These tools are based on the Unix concepts of pipes and filters, and operate on images in much the same way as the standard Unix tools operate on textual data. The Toolkit uses a special run length encoding (RLE) format for storing images and interfacing between the various programs. This reduces the disk space requirements for picture storage and provides a standard header containing descriptive information about an image. Some of the tools are able to work directly with the compressed picture data, increasing their efficiency. A library of C routines is provided for reading and writing the RLE image format, making the toolkit easy to extend.

This paper describes the individual tools, and gives several examples of their use and how they work together. Additional topics that arise in combining images, such as how to combine color table information from multiple sources, are also discussed.

## 1. Introduction

Over the past several years, the University of Utah Computer Graphics Lab has developed several tools and techniques for generating, manipulating and storing images. At first this was done in a somewhat haphazard manner - programs would generate and store images in various formats (often dependent on particular hardware) and data was often not easily interchanged between them. More recently, some effort has been made to standardize the image format, develop an organized set of tools for operating on the images, and develop a subroutine library to make extending this set of tools easier. This paper is about the result of this effort, which we call the *Utah Raster Toolkit*. Using a single efficient image format, the toolkit provides a number of programs for performing common operations on images. These are in turn based on a subroutine library for reading and writing the images.

## 2. Origins

The idea for the toolkit arose while we were developing an image compositor. The compositor (described in detail in section 4.1) allows images to be combined in various ways. We found that a number of simple and independent operations were frequently needed before images could be composited together.

With the common image format, the subroutine library for manipulating it, and the need for a number of independent tools, the idea of a "toolkit" of image manipulating programs arose. These tools are combined using the

---

<sup>1</sup>Originally presented at the third Usenix Workshop on Graphics, Monterey California, November 1986

Unix shell, operating on images much like the standard Unix programs operate on textual data. For example, a Unix user would probably use the sequence of commands:

```
cat /etc/passwd | grep Smith | sort -t: +4.0 | lpr
```

to print a sorted list of all users named "Smith" in the system password file. Similarly, the Raster Toolkit user might do something like:

```
cat image.rle | avg4 | repos -p 0 200 | getfb
```

to downfilter an image and place it on top of the frame buffer screen. The idea is similar to a method developed by Duff [3] for three dimensional rendering.

### 3. The RLE format

The basis of all of these tools is a Run Length Encoded image format [8]. This format is designed to provide an efficient, device independent means of storing multi-level raster images. It is not designed for binary (bitmap) images. It is built on several basic concepts. The central concept is the *channel*. A channel corresponds to a single color, thus there are normally separate red, green and blue channels. Up to 255 color channels are available for use; one channel is reserved for coverage ("alpha") data. Although the format supports arbitrarily deep channels, the current implementation is restricted to 8 bits per channel. An RLE file is treated as a byte stream, making it independent of host byte ordering.

Image data is stored in an RLE file in a scanline form, with the data for each channel of the scanline grouped together. Runs of identical pixel values are compressed into a count and a value. However, sequences of differing pixels are also stored efficiently (i.e, not as a sequence of single pixel runs).

#### 3.1. The RLE header

The file header contains a large amount of information about the image. This includes:

- The size and position on the screen,
- The number of channels saved and the number of bits per channel (currently, only eight bits per channel is supported),
- Several flags, indicating: how the background should be handled, whether or not an alpha channel was saved, if picture comments were saved,
- The size and number of channels in the color map, (if the color map is supplied),
- An optional background color,
- An optional color map,
- An optional set of comments. The comment block contains any number of null-terminated text strings. These strings are conventionally of the form "name=value", allowing for easy retrieval of specific information.

#### 3.2. The scanline data

The scanline is the basic unit that programs read and write. It consists of a sequence of operations, such as *Run*, *SetChannel*, and *Pixels*, describing the actual image. An image is stored starting at the lower left corner and proceeding upwards in order of increasing scanline number. Each operation and its associated data takes up an even number of bytes, so that all operations begin on a 16 bit boundary. This makes the implementation more efficient on many architectures.

Each operation is identified by an 8 bit opcode, and may have one or more operands. Single operand operations fit into a single 16 bit word if the operand value is less than 256. So that operand values are not limited to the range 0..255, each operation has a *long* variant, in which the byte following the opcode is ignored and the following word is taken as a 16 bit quantity. The long variant of an opcode is indicated by setting the bit 0x40 in the opcode (this allows for 64 opcodes, of which 6 have been used so far.)

The current set of opcodes include:

SkipLines	Increment the <i>scanline number</i> by the operand value, terminating the current scanline.
SetColor	Set the <i>current channel</i> to the operand value.
SkipPixels	Skip over pixels in the current scanline. Pixels skipped will be left in the background color.
PixelData	Following this opcode is a sequence of pixel values. The length of the sequence is given by the operand value.
Run	This is the only two operand opcode. The first operand is the length ( $N$ ) of the run. The second operand is the pixel value, followed by a filler byte if necessary <sup>2</sup> . The next $N$ pixels in the scanline are set to the given pixel value.
EOF	This opcode has no operand, and indicates the end of the RLE file. It is provided so RLE files may be concatenated together and still be correctly interpreted. It is not required, a physical end of file also indicates the end of the RLE data.

### 3.3. Subroutine Interface

Two similar subroutine interfaces are provided for reading and writing files in the RLE format. Both read or write a scanline worth of data at a time. A simple "row" interface communicates in terms of arrays of pixel values. It is simple to use, but slower than the "raw" interface, which uses arrays of "opcode" values as its communication medium.

In both cases, the interface must be initialized by calling a setup function. The two types of calls may be interleaved; for example, in a rendering program, the background could be written using the "raw" interface, while scanlines containing image data could be converted with the "row" interface. The package allows multiple RLE streams to be open simultaneously, as is necessary for use in a compositing tool, for example. All data relevant to a particular RLE stream is contained in a "globals" structure. This structure essentially echoes the information in the RLE header, along with current state information about the RLE stream.

## 4. The tools

### 4.1. The image compositor - Comp

**Comp** implements an image compositor based on the compositing algorithms presented in [6]. The compositing operations are based on the presence of an alpha channel in the image. This extra channel usually defines a mask which represents a sort of a cookie-cutter for the image. This is the case when alpha is 255 (full coverage) for pixels inside the shape, zero outside, and between zero and 255 on the boundary. When the compositor operates on images of the cookie-cutter style, the operations behave as follows:

<b>A over B</b> (the default)	The result will be the union of the two image shapes, with A obscuring B in the region of overlap.
<b>A atop B</b>	The result shape is the same as image B, with A obscuring B where the image shapes overlap. (Note that this differs from "over" because the portion of A outside the shape of B will not be in the result image.)
<b>A in B</b>	The result is simply the image A cut by the shape of B. None of the image data of B will be in the result.
<b>A out B</b>	The result image is image A with the shape of B cut out.
<b>A xor B</b>	The result is the image data from both images that is outside the overlap region. The overlap region will be blank.
<b>A plus B</b>	The result is just the sum of the image data. This operation is actually independent of the

---

<sup>2</sup>E.g., a 16 bit pixel value would not need a filler byte.

alpha channels.

The alpha channel can also represent a semi-transparent mask for the image. It would be similar to the cookie-cutter mask, except the interior of the shape would have alpha values that represent partial coverage; e.g. 128 is half coverage. When one of the images to be composited is a semi-transparent mask, the following operations have useful results:

**Semi-transparent A over B**

The image data of B will be blended with that of A in the semi-transparent overlap region. The resulting alpha channel is as transparent as that of image B.

**A in Semi-transparent B**

The image data of A is scaled by the mask of B in the overlap region. The alpha channel is the same as the semi-transparent mask of B.

If the picture to be composited doesn't have an alpha channel present, comp assumes an alpha of 255 (i.e., full coverage) for non-background pixels and an alpha of zero for background pixels.

Comp is able to take advantage of the size information for the two images being composited. For example, if a small picture is being composited over a large backdrop, the actual compositing arithmetic is only performed on a small portion of the image. By looking at the image size in the RLE header, comp performs the compositing operation only where the images overlap. For the rest of the image the backdrop is copied (or not copied, depending on the compositing operation). The "raw" RLE read and write routines are used to perform this copy operation, thus avoiding the cost of compressing and expanding the backdrop as well.

## 4.2. Basic image composition - Repos and Crop

**Repos** and **crop** are the basic tools for positioning and arranging images to be fed to the compositor. **Crop** simply throws away all parts of the image falling outside the rectangle specified. **Repos** positions an image to a specific location on the screen, or moves it by an incremental amount. **Repos** does not have to modify any of the image data, it simply changes the position specification in the RLE header. In order to simplify the code, the Raster Toolkit does not allow negative pixel coordinates in images.

## 4.3. Changing image orientation and size - Flip, Fant and Avg4

Two tools exist for changing the orientation of an image on the screen. **Flip** rotates an image by 90 degrees right or left, turns an image upside down, or reverse it from right to left. **Fant** rotates an image an arbitrary number of degrees from -45 to 45. In order to get rotation beyond -45 to 45, flip and fant can be combined, for example:

```
flip -r < upright.rle | fant -a 10 > rotated.rle
```

rotates an image 100 degrees. **Fant** is also able to scale images by an arbitrary amount in X and Y. A common use for this is to stretch or shrink an image to correct for the aspect ratio of a particular frame buffer. Many frame buffers designed for use with standard video hardware display a picture of 512x480 pixels on a screen with the proportions 4:3, resulting in an overall pixel aspect ratio of 6:5. If the picture **kloo.rle** is digitized or computed assuming a 1:1 aspect ratio (square pixels), the command:

```
cat kloo.rle | fant -s 1.0 1.2 | getfb
```

correctly displays the image on a frame buffer with a 6:5 aspect ratio. **Fant** is implemented using a two-pass subpixel sampling algorithm [4]. This algorithm performs the spatial transform (rotate and/or scale) first on row by row basis, then on a column by column basis. Because the transformation is done on a subpixel level, **fant** does not introduce aliasing artifacts into the image.

**Avg4** downfilters an RLE image into a resulting image of 1/4th the size, by simply averaging four pixel values in the input image to produce a single pixel in the output. If the original image does not contain an alpha channel, **avg4** creates one by counting the number of non-zero pixels in each group of four input pixels and using the count to produce a coverage value. While the alpha channel produced this way is crude (only four levels of coverage) it is enough to make a noticeable improvement in the edges of composited images. One use for **avg4** is to provide anti-aliasing for rendering programs that perform no anti-aliasing of their own. For example, suppose **huge.rle** is a 4k x 4k pixel image rendered without anti aliasing and without an alpha channel. Executing the commands:

```
cat huge.rle | avg4 | avg4 | avg4 > small.rle
```

produces an image **small.rle** with 64 (8x8) samples per pixel and an alpha channel with smooth edges.

Images generated from this approach are as good as those produced by direct anti-aliasing algorithms such as the A-buffer [2]. However, a properly implemented A-buffer renderer produces images nearly an order of magnitude faster.

#### 4.4. Color map manipulation - Ldmap and Applymap

As mentioned previously, RLE files may optionally contain a color map for the image. **Ldmap** is used to create or modify a color map within an RLE file. Ldmap is able to create some standard color maps, such as linear maps with various ramps, or maps with various gamma corrections. Color maps may also be read from a simple text file format, or taken from other RLE files. Ldmap also performs *map composition*, where one color map is used as an index into the other. An example use for this is to apply a gamma correction to an image already having a non-linear map.

**Applymap** applies the color map in an rle file to the pixel values in the file. For example, if the color map in **kloo.rle** contained a color map with the entries (for the red channel):

Index	Red color map
0:	5
1:	7
2:	9

Then a (red channel) pixel value of zero would be displayed with an intensity of five (assuming the display program used the color map in **kloo.rle**). When **kloo.rle** is passed through **applymap**,

```
cat kloo.rle | applymap > kloo2.rle
```

pixels that had a value of zero in **kloo.rle** now have a value of five in **kloo2.rle**, pixel values of one would now be seven, etc. When displaying the images on a frame buffer, **kloo2.rle** appears the same with a linear map loaded as **kloo.rle** does with its special color map loaded.

One use for these tools is merging images with different compensation tables. For example, suppose image **gam.rle** was computed so that it requires a gamma corrected color table (stored in the RLE file) to be loaded to look correct on a particular monitor, and image **lin.rle** was computed with the gamma correction already taken into consideration (so it looks correct with a linear color table loaded). If these two images are composited together without taking into consideration these differences, the results aren't correct (part of the resulting image is either too dim or washed out). However, we can use **applymap** to "normalize" the two images to using a linear map with:

```
cat gam.rle | applymap | comp - lin.rle | ldmap -l > result.rle
```

#### 4.5. Generating backgrounds

Unlike most of the toolkit programs which act as filters, **background** just produces output. Background either produces a simple flat field, or it produces a field with pixel intensities ramped in the vertical direction. Most often background is used simply to provide a colored backdrop for an image. Background is another example of a program that takes advantage of the "raw" RLE facilities. Rather than generating the scanlines and compressing them, background simply generates the opcodes required to produce each scanline.

#### 4.6. Converting full RGB images to eight bits - to8 and tobw

Although most of the time we prefer to work with full 24 bit per pixel images, (32 bits with alpha) we often need the images represented with eight bits per pixel. Many inexpensive frame buffers (such as the AED 512) and many personal color workstations (VaxStation GPX, Color Apollos and Suns) are equipped with eight bit displays.

**To8** converts a 24 bit image to an eight bit image by applying a dither matrix to the pixels. This basically trades spatial resolution for color resolution. The resulting image has eight bits of data per pixel, and also contains a special color map for displaying the dithered image (since most eight bit frame buffers still use 24 bit wide color table entries).



**Tobw** converts a picture from 24 bits of red, green, and blue to an eight bit gray level image. It uses the standard television YIQ transformation of

$$graylevel = 0.35 \times red + 0.55 \times green + 0.10 \times blue.$$

These images are often preferred when displaying the image on an eight bit frame buffer and shading information is more important than color.

## 5. Interfacing to the RLE toolkit

In order to display RLE images, a number of programs are provided for displaying the pictures on various devices. These display programs all read from standard input, so they are conveniently used as the end of a raster toolkit pipeline. The original display program, **getfb**, displays images on our ancient Grinnell GMR-27 frame buffer. Many display programs have since been developed:

<b>getcxc</b>	displays images on a Chromatics CX1500
<b>getiris</b>	displays an image on an Iris workstation (via ethernet)
<b>getX</b>	for the X window system
<b>getap</b>	for the Apollo display manager
<b>gethp</b>	for the Hewlett-Packard Series 300 workstation color display
<b>rletops</b>	converts an RLE file to gray-level PostScript output <sup>3</sup>

The **getX**, **getap** and **gethp** programs automatically perform the dithering required to convert a 24 bit RLE image to eight bits (for color nodes) or one bit (for bitmapped workstations). Since all of these workstations have high resolution (1Kx1K) displays, the trade of spatial resolution for color resolution produces very acceptable results. Even on bitmapped displays the image quality is good enough to get a reasonable idea of an image's appearance.

Two programs, **painttorle** and **rletopaint** are supplied to convert MacPaint images to RLE files. This offers a simple way to add text or graphic annotation to an RLE image (although the resolution is somewhat low).

## 6. Examples

A typical use for the toolkit is to take an image generated with a rendering program, add a background to it, and display the result on a frame buffer:

```
rlebg 250 250 250 -v | comp image.rle - | getfb
```

What follows are some more elaborate applications:

### 6.1. Making fake shadows

In this exercise we take the dart (Figure 6-1a) and stick it into the infamous mandrill, making a nice (but completely fake) shadow along the way.

First the image of the dart is rotated by 90 degrees (using **rleflip -l** and then stretched in the X direction (using **fant -s 1.3 1.0**) to another image we can use as a shadow template, figure 6-1b. Then we take this image, **dart\_stretch\_rot.rle**, and do:

```
rlebg 0 0 0 175 \  
| comp -o in - dart_stretch_rot.rle > dart_shadow.rle
```

This operation uses the stretched dart as a cookie-cutter, and the shape is cut out of a black image, with a coverage mask (alpha channel) of 175. Thus when we composite the shadow (figure 6-2a) over the mandrill image, it lets 32% of the mandrill through  $((255 - 175) / 255 = 32\%)$  with the rest being black.

---

<sup>3</sup>**Rletops** was used to produce the figures in this paper.



**Figure 6-1:** **a:** Original dart image. **b:** Rotated and stretched dart.



**Figure 6-2:** **a:** Dart shadow mask. **b:** Resulting skewered baboon.

Now all we have left to do is to composite the dart and the shadow over the mandrill image, and the result is figure 6-2b. Note that since the original dart image was properly anti-aliased, no aliasing artifacts were introduced in the final image.

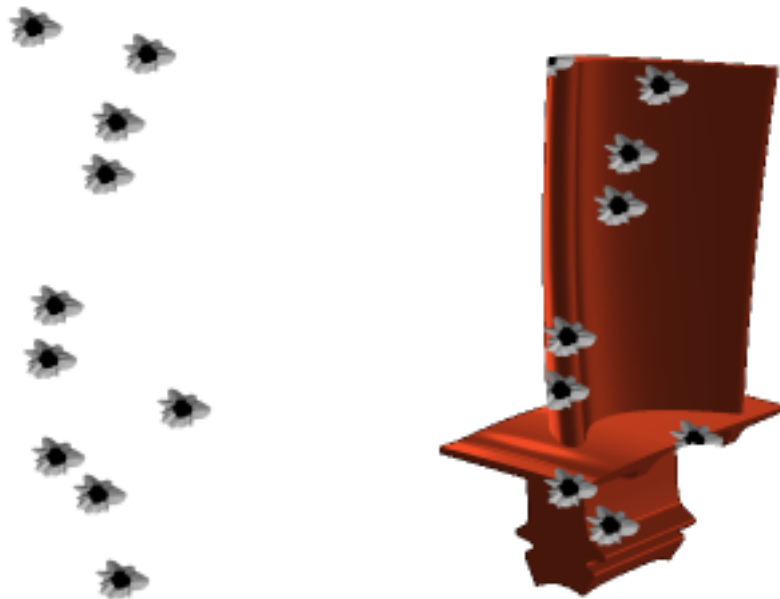
## 6.2. Cutting holes

In this example we start out with a single bullet hole, created with the same modelling package that produced the dart.<sup>4</sup> First the hole (which originally filled the screen) is downfiltered to a small size, with

---

<sup>4</sup>The bullet hole was modeled with cubic B-splines. Normal people probably would have painted something like this...

```
avg4 < big_hole.rle | avg4 | avg4 > smallhole.rle
```

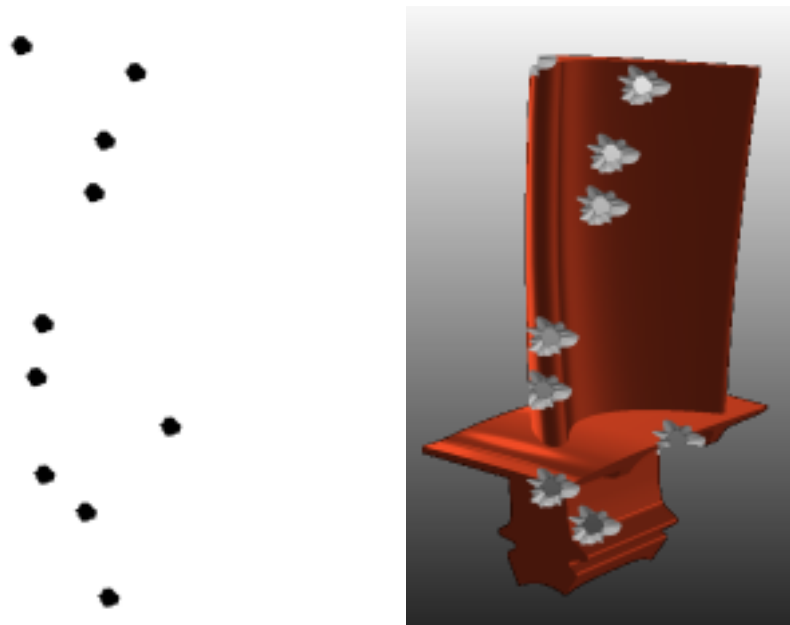


**Figure 6-3:** **a:** A set of bullet holes. **b:** A shot-up turbine blade.

Now by invoking **repos** and **comp** several times in a row, a set of shots is built up (Figure 6-3a)

To shoot up the turbine blade, the *atop* operator of **comp** is used. This works much like *over*, except the bullet holes outside of the turbine blade don't appear in the resulting image (Figure 6-3b) Note how the top left corner is just grazed.

But there's still one catch. Suppose we want to display the shot-up turbine blade over a background of some sort. We want to be able to see the background through the holes. To do this we come up with another mask to cut away the centers of the bullet holes already on the turbine blade so we can see through them (Figure 6-4a).

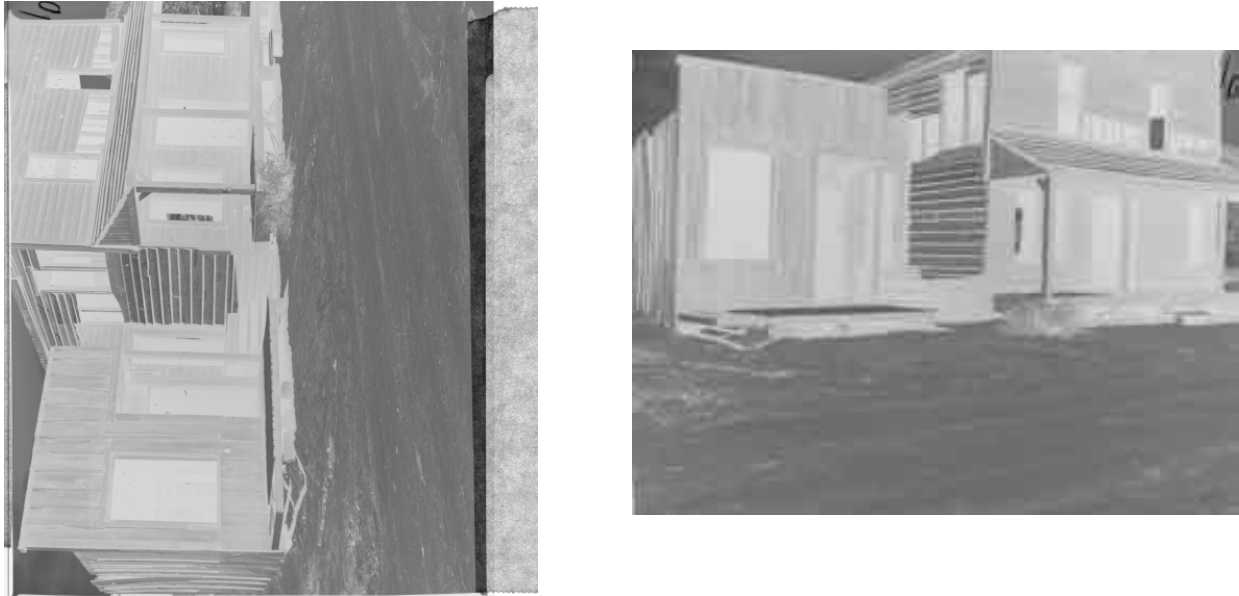


**Figure 6-4:** **a:** Cut-away masks for the bullet hole centers.  
**b:** Finished turbine blade

Now we can see through the blade (Figure 6-4b).

### 6.3. Integrating digitized data

In this example, we take a raw digitized negative and turn it into a useful frame buffer image. Figure 6-5a shows the original image from the scanner. First the image is cropped to remove the excess digitized portion, then rotated into place with **flip -r**, resulting in Figure 6-5b.



**Figure 6-5:** a: The raw digitized negative. b: Cropped and rotated image

To get the image to appear as a positive on the frame buffer, a negative linear color map (stored as text file) is loaded, then this is composed with a gamma correction map of 2.2, and finally applied to the pixels with the command:

```
ldmap -f neg.cmap < neg_pahriah.rle | ldmap -a -g 2.2 \
| applymap > pahriah.rle
```

The result, Figure 6-6 now appears correct with a linear color map loaded.

## 7. Future Work

Needless to say, a project such as the Raster Toolkit is open ended in nature, and new tools are easily added. For example, most of the tools we have developed to date deal with image synthesis and composition, primarily because that is our research orientation. Additional tools could be added to assist work in areas such as vision research or image processing (for examples, see [1, 7]).

Another interesting application would be a visual "shell" for invoking the tools. Currently, arguments to programs like **crop** and **repos** are specified by tediously finding the numbers with the frame buffer cursor and then typing them into a shell. The visual shell would allow the various toolkit operations to be interactively selected from a menu. Such a shell could probably work directly with an existing workstation window system such as X [5] to provide a friendly environment for using the toolkit.

## 8. Conclusions

The Raster Toolkit provides a flexible, simple and easily extended set of tools for anybody working with images in a Unix-based environment. We have ported the toolkit to a wide variety of Unix systems, including Gould UTX, HP-UX, Sun Unix, Apollo Domain/IX, 4.2 and 4.3 BSD, etc. The common interfaces of the RLE format and the subroutine library make it easy to interface the toolkit to a wide variety of image sources and displays.



**Figure 6-6:** Final image of the old Pahriah ghost town

## **9. Acknowledgments**

This work was supported in part by the National Science Foundation (DCR-8203692 and DCR-8121750), the Defense Advanced Research Projects Agency (DAAK11-84-K-0017), the Army Research Office (DAAG29-81-K-0111), and the Office of Naval Research (N00014-82-K-0351). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

## References

- [1] Burt, Peter J, and Adelson, Edward H.  
A Multiresolution Spline With Application to Image Mosaics.  
*ACM Transactions on Graphics* 2(4):217-236, October, 1983.
- [2] Carpenter, Loren.  
The A-Buffer, An Antialiased Hidden Surface Method.  
*Computer Graphics* 18(3):103, July, 1984.  
Proceedings of SIGGRAPH 84.
- [3] Duff, Tom.  
Compositing 3-D Rendered Images.  
*Computer Graphics* 19(3):41, July, 1985.  
Proceedings of SIGGRAPH 85.
- [4] Fant, Karl M.  
A Nonaliasing, Real-Time, Spatial Transform.  
*IEEE Computer Graphics and Applications* 6(1):71, January, 1986.
- [5] Gettys, Jim, Newman, Ron, and Fera, Tony D.  
*Xlib - C Language X Interface, Protocol Version 10*.  
Technical Report, MIT Project Athena, January, 1986.
- [6] Porter, Thomas and Duff, Tom.  
Compositing Digital Images.  
*Computer Graphics* 18(3):253, July, 1984.  
Proceedings of SIGGRAPH 84.
- [7] Stockham, Thomas G.  
Image Processing in the Context of a Visual Model.  
*Proceedings of the IEEE* 60(7):828-842, July, 1972.
- [8] Thomas, Spencer W.  
*Design of the Utah RLE Format*.  
Technical Report 86-15, Alpha\_1 Project, CS Department, University of Utah, November, 1986.

# Design of the Utah RLE Format

Spencer W. Thomas

University of Utah, Department of Computer Science

## Abstract

The Utah RLE (Run Length Encoded) format is designed to provide an efficient, device independent means of storing multi-level raster images. Images of arbitrary size and depth can be saved. The design of the format is presented, followed by descriptions of the library routines used to create and read RLE format files.

## 1. Introduction

The Utah RLE (Run Length Encoded) format is designed to provide an efficient, device independent means of storing multi-level raster images. It is not designed for binary (bitmap) images. It is built on several basic concepts. The central concept is that of a *channel*. A channel corresponds to a single color, thus there are normally a red channel, a green channel, and a blue channel. Up to 255 color channels are available for use; one channel is reserved for "alpha" data. Although the format supports arbitrarily deep channels, the current implementation is restricted to 8 bits per channel.

Image data is stored in an RLE file in a scanline form, with the data for each channel of the scanline grouped together. Runs of identical pixel values are compressed into a count and a value. However, sequences of differing pixels are also stored efficiently (not as a sequence of single pixel runs).

The file header contains a large amount of information about the image, including its size, the number of channels saved, whether it has an alpha channel, an optional color map, and comments. The comments may be used to add arbitrary extra information to the saved image.

A subroutine interface has been written to allow programs to read and write files in the RLE format. Two interfaces are available, one that completely interprets the RLE file and returns scanline pixel data, and one that returns a list of "raw" run and pixel data. The second is more efficient, but more difficult to use, the first is easy to use, but slower.

The Utah RLE format has been used to save images from many sources, and to display saved images on many different displays and from many different computers.

## 2. Description of RLE Format

All data in the RLE file is treated as a byte stream. Where quantities larger than 8 bits occur, they are written in PDP-11 byte order (low order byte first).

The RLE file consists of two parts, a header followed by scanline data. The header contains general information about the image, while the scanline data is a stream of operations describing the image itself.



## 2.1. The Header

Magic number	
xpos	
ypos	
xsize	
ysize	
flags	ncolors
pixelbytes	ncmap
cmaplen	red bg
green bg	blue bg
color map entry 0	
color map entry 1	
I	

Figure 2-1: RLE file header

The header has a fixed part and a variable part. A diagram of the header is shown in Figure 2-1. The magic number identifies the file as an RLE file. Following this are the coordinates of the lower left corner of the image and the size of the image in the X and Y directions. Images are defined in a first quadrant coordinate system (origin at the lower left, X increasing to the right, Y increasing up.) Thus, the image is enclosed in the rectangle

$[xpos, xpos+xsize-1] \times [ypos, ypos+ysize-1]$ .

The position and size are 16 bit integer quantities; images up to 32K square may be saved (the sizes should not be negative).

A flags byte follows. There are currently four flags defined:

ClearFirst	If this flag is set, the image rectangle should first be cleared to the background color (q.v.) before reading the scanline data.
NoBackground	If this flag is set, no background color is supplied, and the ClearFirst flag should be ignored.
Alpha	This flag indicates the presence of an "alpha" channel. The alpha channel is used by image compositing software to correctly blend anti-aliased edges. It is stored as channel -1 (255).
Comments	If this flag is set, comments are present in the variable part of the header, immediately following the color map.

The next byte is treated as an unsigned 8 bit value, and indicates the number of color channels that were saved. It may have any value from 0 to 254 (channel 255 is reserved for alpha values).

The *pixelbits* byte gives the number of bits in each pixel. The only value currently supported by the software is 8 (in fact, this byte is currently ignored when reading RLE files). Pixel sizes taking more than one byte will be packed low order byte first.

The next two bytes describe the size and shape of the color map. *Ncmap* is the number of color channels in the color map. It need not be identical to *ncolors*, but interpretation of values of *ncmap* different from 0, 1, or *ncolors* may be ambiguous, unless *ncolors* is 1. If *ncmap* is zero, no color map is saved. *Cmaplen* is the log base 2 of the

length of each channel of the color map. Thus, a value for *cmaplen* of 8 indicates a color map with 256 entries per channel.

Immediately following the fixed header is the variable part of the file header. It starts with the background color. The background color has *ncolors* entries; if necessary, it is filled out to an odd number of bytes with a filler byte on the end (since the fixed header is an odd number bytes long, this returns to a 16 bit boundary).

Following the background color is the color map, if present. Color map values are stored as 16 bit quantities, left justified in the word. Software interpreting the color map must apply a shift appropriate to the application or to the hardware being used. This convention permits use of the color map without knowing the original output precision. The channels of the map are stored in increasing numerical order (starting with channel 0), with the entries of each channel stored also in increasing order (starting with entry 0). The color map entries for each channel are stored contiguously.

Comments, if present, follow the color map. A 16 bit quantity giving the length of the comment block comes first. If the length is odd, a filler byte will be present at the end, restoring the 16 bit alignment (but this byte is not part of the comments). The comment block contains any number of null-terminated text strings. These strings will conventionally be of the form "name=value", allowing for easy retrieval of specific information. However, there is no restriction that a given name appear only once, and a comment may contain an arbitrary string. The intent of the comment block is to allow information to be attached to the file that is not specifically provided for in the RLE format.

## 2.2. The Scanline Data

The scanline data consists of a sequence of operations, such as *Run*, *SetChannel*, and *Pixels*, describing the actual image. An image is stored starting at the lower left corner and proceeding upwards in order of increasing scanline number. Each operation and its associated data takes up an even number of bytes, so that all operations begin on a 16 bit boundary. This makes the implementation more efficient on many architectures.

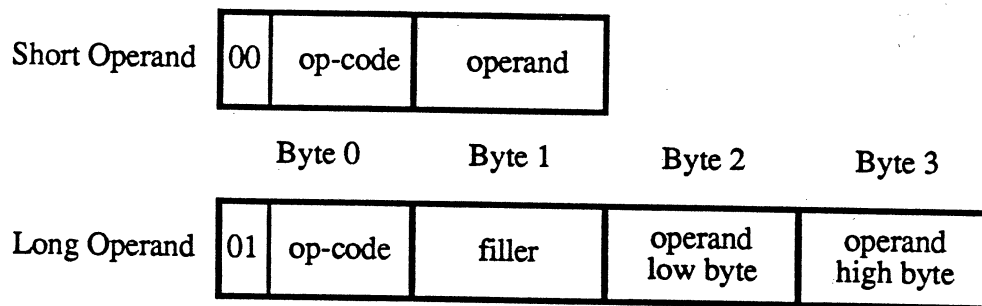


Figure 2-2: RLE file operand formats

Each operation is identified by an 8 bit opcode, and may have one or more operands. Single operand operations fit into a single 16 bit word if the operand value is less than 256. So that operand values are not limited to the range 0..255, each operation has a *long* variant, in which the byte following the opcode is ignored and the following word is taken as a 16 bit quantity. The long variant of an opcode is indicated by setting the bit 0x40 in the opcode (this allows for 64 opcodes, of which 6 have been used so far.) The two single operand formats are shown pictorially in Figure 2-2.

The individual operations will now be discussed in detail. The descriptions are phrased in terms of the actions necessary to interpret the file. Three indices are necessary: the *current channel*, the *scanline number*, and the *pixel index*. The *current channel* is the channel to which data operations apply. The *scanline number* is just the Y position of the scanline in the image. The *pixel index* is the X position of the pixel within the scanline. The operations are:

SkipLines	Increment the <i>scanline number</i> by the operand value. This operation terminates the current scanline. The <i>pixel index</i> should be reset to the <i>xpos</i> value from the header.
SetColor	Set the <i>current channel</i> to the operand value. This operation does not have a long variant. Note that an operand value of 255 will be interpreted as a -1, indicating the alpha channel. All other operand values are positive. The <i>pixel index</i> is reset to the <i>xpos</i> value.
SkipPixels	Skip over pixels in the current scanline. Increment <i>pixel index</i> by the operand value. Pixels skipped will be left in the background color.
PixelData	Following this opcode is a sequence of pixel values. The length of the sequence is given by the operand value. If the length of the sequence is odd, a filler byte is appended. Pixel values are inserted into the scanline in increasing X order. The <i>pixel index</i> is incremented by the sequence length.
Run	This is the only two operand opcode. The first operand is the length ( <i>N</i> ) of the run. The second operand is the pixel value, followed by a filler byte if necessary <sup>1</sup> . The next <i>N</i> pixels in the scanline are set to the given pixel value. The <i>pixel index</i> is incremented by <i>N</i> , to point to the pixel following the run.
EOF	This opcode has no operand, and indicates the end of the RLE file. It is provided so that RLE files may be concatenated together and still be correctly interpreted. It is not required, a physical end of file will also indicate the end of the RLE data.

### 2.3. Subroutine Interface

Two similar subroutine interfaces are provided for reading and writing files in the RLE format. Both read or write a scanline worth of data at a time. A simple "row" interface communicates in terms of arrays of pixel values. It is simple to use, but slower than the "raw" interface, which uses a list of "opcode" values as its communication medium.

In both cases, the interface must be initialized by calling a setup function. The two types of calls may be interleaved; for example, in a rendering program, the background could be written using the "raw" interface, while scanlines containing image data could be converted with the "row" interface. The package allows multiple RLE streams to be open simultaneously, as is necessary for use in a compositing tool, for example. All data relevant to a particular RLE stream is contained in a "globals" structure.

The globals structure echoes the format of the RLE header. The fields are described below:

dispatch	The RLE creation routines are capable of writing various types of output files in addition to RLE. This value is an index into a dispatch table. This value is initialized by <i>sv_setup</i> .
ncolors	The number of color channels in the output file. Up to this many color channels will be saved, depending on the values in the channel bitmap (see below).
bg_color	A pointer to an array of <i>ncolors</i> integers containing the background color.
alpha	If this is non-zero, an alpha channel will be saved. The presence or absence of an alpha channel has no effect on the value in <i>ncolors</i> .
background	Indicates how to treat background pixels. It has the following values: <ul style="list-style-type: none"> <li>0 Save all pixels, the background color is ignored.</li> <li>1 Save only non-background pixels, but don't set the "clear screen" bit. This indicates "overlay" mode, a cheap form of compositing (but see note below about this.)</li> <li>2 Save only non-background pixels, clear the screen to the background color before restoring the image.</li> </ul>

<sup>1</sup>E.g., a 16 bit pixel value would not need a filler byte.

<code>xmin, xmax, ymin, ymax</code>	Inclusive bounds of the image region being saved.
<code>ncmap</code>	Number of channels of color map to be saved. The color map will not be saved if this is 0.
<code>cmaplen</code>	Log base 2 of the number of entries in each channel of the color map.
<code>cmap</code>	Pointer to an array containing the color map. The map is saved in "channel major" order. Each entry in the map is a 16 bit value with the color value left justified in the word. If this pointer is NULL, no color map will be saved.
<code>comments</code>	Pointer to an array of pointers to strings. The array is terminated by a NULL pointer (like <i>argv</i> or <i>envp</i> ). If this pointer is NULL or if the first pointer it points to is NULL, comments will not be saved.
<code>fd</code>	File (FILE *) pointer to be used for writing or reading the RLE file.
<code>bits</code>	A bitmap containing 256 bits. A channel will be saved (or retrieved) only if the corresponding bit is set in the bitmap. The alpha channel corresponds to bit 255. The bitmap allows an application to easily ignore color channel data that is irrelevant to it.

The `globals` structure also contains private data for use by the RLE reading and writing routines; data that must be maintained between calls, but that applies to each stream separately.

## 2.4. Writing RLE files

To create a run-length encoded file, one first initializes a `globals` structure with the relevant information about the image, including the output file descriptor. The output file should be open and empty. Then one calls `sv_setup`:

```
sv_setup( RUN_DISPATCH, &globals );
```

This writes the file header and initializes the private portions of the global data structure for use by the RLE file writing routines.

The image data must be available or expressible in a scanline order (with the origin at the bottom of the screen). After each scanline is computed, it is written to the output file by calling one of `sv_putrow` or `sv_putraw`. If a vertical interval of the image has no data, it may be skipped by calling `sv_skiprow`:

```
/* Skip nrow scanlines */  
sv_skiprow( &globals, nrow );
```

If the image data for a scanline is available as an array of pixel values, `sv_putrow` should be used to write the data to the output file. As an example, let us assume that we have a 512 pixel long scanline, with three color channels and no alpha data. We could call `sv_putrow` as follows:

```
rle_pixel scandata[3][512], *rows[3];  
int i;  
  
for ( i = 0; i < 3; i++ )  
    rows[i] = scandata[i];  
sv_putrow( rows, 512, &globals );
```

Note that `sv_putrow` is passed an array of pointers to vectors of pixels. This makes it easy to pass arbitrarily many, and to specify values of `rowlen` different from the size of (e.g.) the `scandata` array.

The first element of each row of pixels is the pixel at the `xmin` location in the scanline. Therefore, when saving only part of an image, one must be careful to set the `rows` pointers to point to the correct pixel in the scanline.

If an alpha channel is specified to be saved, things get a little more complex. Here is the same example, but now with an alpha channel being saved.

```

rle_pixel scandata[3][512],
        alpha[512], *rows[4];
int i;

rows[0] = alpha;
for ( i = 0; i < 3; i++ )
    rows[i+1] = scandata[i];
sv_putrow( rows+1, 512, &globals );

```

The *sv\_putrow* routine expects to find the pointer to the alpha channel at the -1 position in the rows array. Thus, we pass a pointer to *rows[1]* and put the pointer to the alpha channel in *rows[0]*.

Finally, after all scanlines have been written, we call *sv\_puteof* to write an EOF opcode into the file. This is not strictly necessary, since a physical end of file also indicates the end of the RLE data, but it is a good idea.

Here is a skeleton of an application that uses *sv\_putrow* to save an image is shown in Figure 2-3. This example uses the default values supplied in the globals variable *sv\_globals*, modifying it to indicate the presence of an alpha channel.

Using *sv\_putrow* is more complicated, as it takes arrays of *rle\_op* structures instead of just pixels. If the data is already available in something close to this form, however, *sv\_putrow* will run much more quickly than *sv\_putrow*. An *rle\_op* is a structure with the following contents:

opcode	The type of data. One of <i>ByteData</i> or <i>RunData</i> .
xloc	The X location within the scanline at which this data begins.
length	The length of the data. This is either the number of pixels that are the same color, for a run, or the number of pixels provided as byte data.
pixels	A pointer to an array of pixel values. This field is used only for the <i>ByteData</i> opcode.
run_val	The pixel value for a <i>RunData</i> opcode.

Since there is no guarantee that the different color channels will require the same set of *rle\_ops* to describe their data, a separate count must be provided for each channel. Here is a sample call to *sv\_putrow*:

```

int nraw[3];      /* Length of each row */
rle_op *rows[3]; /* Data pointers */
sv_putrow( rows, nraw, &globals );

```

A more complete example of the use of *sv\_putrow* will be given in connection with the description of *rle\_getrow*, below.

Calls to *sv\_putrow* and *sv\_putrow* may be freely intermixed, as required by the application.

## 2.5. Reading RLE Files

Reading an RLE file is much like writing one. An initial call to a setup routine reads the file header and fills in the *globals* structure. Then, a scanline at a time is read by calling *rle\_getrow* or *rle\_getrow*.

The calling program is responsible for opening the input file. A call to *rle\_get\_setup* will then read the header information and fill in the supplied globals structure. The return code from *rle\_get\_setup* indicates a variety of errors, such as the input file not being an RLE file, or encountering an EOF while reading the header.

Each time *rle\_getrow* is called, it fills in the supplied scanline buffer with one scanline of image data and returns the Y position of the scanline (which will be one greater than the previous time it was called). Depending on the setting of the *background* flag, the scanline buffer may or may not be cleared to the background color on each call. If it is not (*background* is 0 or 1), and if the caller does not clear the buffer between scanlines, then a "smearing" effect will be seen, if some pixels from previous scanlines are not overwritten by pixels on the current scanline. Note that if *background* is 0, then no background color was supplied, and setting *background* to 2 to try to get automatic buffer clearing will usually cause a segmentation fault when *rle\_getrow* tries to get the background color through the *bg\_color* pointer.

```

#include <svfb_global.h>

main()
{
    rle_pixel scanline[3][512], alpha[512], *rows[4];
    int y, i;

    /* Most of the default values in sv_globals are ok */
    /* We do have an alpha channel, though */
    sv_globals.sv_alpha = 1;
    SV_SET_BIT( sv_globals, SV_ALPHA );

    rows[0] = alpha;
    for ( i = 0; i < 3; i++ )
        rows[i+1] = scanline[i];

    sv_setup( RUN_DISPATCH, &sv_globals );

    /* Create output for 512 x 480 (default size) display */
    for ( y = 0; y < 480; y++ )
    {
        mk_scanline( y, scanline, alpha );
        sv_putrow( rows, 512, &sv_globals );
    }
    sv_puteof( &sv_globals );
}

```

Figure 2-3: Example of use of `sv_putrow`

Figure 2-4 shows an example of the use of `rle_getrow`. Note the dynamic allocation of scanline storage space, and compensation for presence of an alpha channel. A subroutine, `rle_row_alloc`, is available that performs the storage allocation automatically. It is described below. If the alpha channel were irrelevant, the macro `SV_CLR_BIT` could be used to inhibit reading it, and no storage space would be needed for it.

The function `rle_getrow` is the inverse of `sv_putrow`. When called, it fills in the supplied buffer with raw data for a single scanline. It returns the scanline `y` position, or  $2^{15}$  to indicate end of file. It is assumed that no image will have more than  $2^{15}-1$  scanlines. A complete program (except for error checking) that reads an RLE file from standard input and produces a negative image on standard output is shown in Figure 2-5.

The functions `rle_row_alloc` and `rle_raw_alloc` simplify allocation of buffer space for use by the rle routines. Both use a supplied globals structure to determine how many and which channels need buffer space, as well as the size of the buffer for each scanline. The returned buffer pointers will be adjusted for the presence of an alpha channel, if it is present. Buffer space for pixel or `rle_op` data will be allocated only for those channels that have bits set in the channel bitmap. The buffer space may be freed by calling `rle_row_free` or `rle_raw_free`, respectively.

### 3. Comments, issues, and directions

Some comments on the file format and current subroutine implementation:

- The background color for the alpha channel is always 0.
- All channels must have the same number of bits. This could be a problem when saving, e.g., Z values, or if more than 8 bits of precision were desired for the alpha channel.
- Pixels are skipped (by `sv_putrow`) only if all channel values of the pixel are equal to the corresponding background color values.
- The current Implementation of `sv_putrow` skips pixels only if at least 2 adjacent pixels are equal to the background. The SkipPixels operation is intended for efficiency, not to provide cheap compositing.

```

/* An example of using rle_getrow */
/* Scanline pointer */
rle_pixel ** scan;
int i;

/* Read the RLE file from stdin */
rle_get_setup( &globals );

/* Allocate enough space for scanline data, including alpha channel */
/* (Should check for non-zero return, indicating a malloc error) */
rle_row_alloc( &globals, &scan );

/* Read scanline data */
while ( (y = rle_getrow( &globals, scan scan ) <= globals.sv_ymax )
        /* Use the scanline data */;

```

Figure 2-4: Example of rle\_getrow use.

- Nothing forces the image data to lie within the bounds declared in the header. However, *rle\_getrow* will not write outside these bounds, to prevent core dumps. No such protection is provided by *rle\_getraw*.
- Images saved in RLE are usually about 1/3 their original size (for an "average" image). Highly complex images may end up slightly larger than they would have been if saved by the trivial method.

We have not yet decided how pixels with other than 8 bits should be packed into the file. To keep the file size down, one would like to pack *ByteData* as tightly as possible. However, for interpretation speed, it would probably be better to save one value in each  $(\text{pixelbits}+7)/8$  bytes.

Some proposed enhancements include:

- A "ramp" opcode. This specifies that pixel values should be linearly ramped between two values for a given number of pixels in the scanline. This opcode would be difficult to generate from an image, but if an application knew it was generating a ramp, it could produce significant file size savings (e.g. in Gouraud shaded images).
- Opcodes indicating that the current scanline is identical to the previous, or that it differs only slightly (presumably followed by standard opcodes indicating the difference). Detection of identical scanlines is easy, deciding that a scanline differs slightly enough to warrant a differential description could be difficult. In images with large areas with little change, this could produce size savings<sup>2</sup>

The subroutine library is still missing some useful functions. Some proposed additions are:

- Conversion from "raw" to "row" format, and back. One could then view *sv\_putrow* as being a "raw" to "row" conversion followed by a call to *sv\_putraw*, and *rle\_getrow* as a call to *rle\_getraw* followed by "row" to "raw" conversion.
- A function to merge several channels of "raw" data into a single channel. For example, this would take separate red, green, and blue channels and combine them into a single RGB channel. This would be useful for RLE interpretation on devices that do not easily support the separate channel paradigm, while preserving the efficiency of the "raw" interface. It could also be used to increase the efficiency of a compositing program.

The Utah RLE format has developed and matured over a period of about six years, and has proven to be versatile and useful for a wide variety of applications that require image transmittal and storage. It provides a compact, efficiently interpreted image storage capability. We expect to see continued development of capabilities and utility, but expect very little change in the basic format.

<sup>2</sup>This suggestion was inspired by a description of the RLE format used at Ohio State University.

```

#include <stdio.h>
#include <svfb_global.h>
#include <rle_getraw.h>

main()
{
    struct sv_globals in_glob, out_glob;
    rle_op ** scan;
    int * nraw, i, j, c, y, newy;

    in_glob.svfb_fd = stdin;
    rle_get_setup( &in_glob );
    /* Copy setup information from input to output file */
    out_glob = in_glob;
    out_glob.svfb_fd = stdout;

    /* Get storage for calling rle_getraw */
    rle_raw_alloc( &in_glob, &scan, &nraw );

    /* Negate background color! */
    if ( in_glob.sv_background )
        for ( i = 0; i < in_glob.sv_ncolors; i++ )
            out_glob.sv_bg_color[i] = 255 - out_glob.sv_bg_color[i];

    /* Init output file */
    sv_setup( RUN_DISPATCH, &out_glob );

    y = in_glob.sv_ymin;
    while ( (newy = rle_getraw( &in_glob, scan, nraw )) != 32768 ) {
        /* If > one line skipped in input, do same in output */
        if ( newy - y > 1 )
            sv_skiprow( &out_glob, newy - y );
        y = newy;
        /* Map all color channels */
        for ( c = 0; c < out_glob.sv_ncolors; c++ )
            for ( i = 0; i < nraw[c]; i++ )
                switch( scan[c][i].opcode ) {
                    case RRunDataOp:
                        scan[c][i].u.run_val = 255 - scan[c][i].u.run_val;
                        break;
                    case RByteDataOp:
                        for ( j = 0; j < scan[c][i].length; j++ )
                            scan[c][i].u.pixels[j] =
                                255 - scan[c][i].u.pixels[j];
                        break;
                }
            sv_putraw( scan, nraw, &out_glob );
        /* Free raw data */
        rle_freeraw( &in_glob, scan, nraw );
    }
    sv_puteof( &out_glob );

    /* Free storage */
    rle_raw_free( &in_glob, scan, nraw );
}

```

Figure 2-5: Program to produce a negative of an image



#### **4. Acknowledgments**

This work was supported in part by the National Science Foundation (DCR-8203692 and DCR-8121750), the Defense Advanced Research Projects Agency (DAAK11-84-K-0017), the Army Research Office (DAAG29-81-K-0111), and the Office of Naval Research (N00014-82-K-0351). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

# **Toolkit User's Guide**



**NAME**

**applymap** - Apply the color map in an RLE file to the pixel data

**SYNOPSIS**

**applymap** [ **-l** ] [ **-o outfile** ] [ *rlefile* ]

**DESCRIPTION**

This program takes the color map in an *RLE(5)* file and modifies the pixel values by applying the color map to them. If there is more than one color channel in the input file, the color map in the input file should have the same number of channels. If the input file has a single color channel, the output file will have the same number of color channels as the color map.

Each pixel in the input file is mapped as follows: For a multi-channel input file, a pixel in channel *i* is mapped as *map[i][pixel] >> 8*, producing a pixel in output channel *i*. The right shift takes the 16 bit color map value to an 8 bit pixel value. For a single channel input file, to produce a pixel in output channel *i* is produced from the corresponding input pixel value as *map[i][pixel] >> 8*.

The options are

**-l** This option will cause a linear (identity) color map to be loaded into the output file. Otherwise, the output file will have no color map.

*rlefile* The input will be read from this file, otherwise, input will be taken from stdin.

**-o outfile**

If specified, output will be written to this file, otherwise it will go to stdout.

**SEE ALSO**

*rleldmap(1)*, *RLE(5)*.

**AUTHOR**

Spencer W. Thomas, University of Utah

**BUGS**

If the image data and color map channels in the input file do not conform to the restriction stated above (*N->N* or *1->N*) the program will most likely core dump.

**NAME**

**avg4** – Downfilter an image by simple averaging.

**SYNOPSIS**

**avg4** [ infile ] > outfile

**DESCRIPTION**

*Avg4* downfilters an RLE image into a resulting image of 1/4th the size, by simply averaging four pixel values in the input image to produce a single pixel in the output. If the original image does not contain an alpha channel, *avg4* creates one by counting the number of non-zero pixels in each group of four input pixels and using the count to produce a coverage value. While the alpha channel produced this way is crude (only four levels of coverage) it is enough to make a noticeable improvement in the edges of composited images.

**SEE ALSO**

*RLE(5)*, *comp(1)*

**AUTHOR**

Rod Bogart, John W. Peterson

**BUGS**

Very simple minded - more elaborate filters could be implemented.

## NAME

comp - Digital image compositor

## SYNOPSIS

comp [ -o op ] Afile Bfile > outfile

## DESCRIPTION

*comp* implements an image compositor based on presence of an alpha, or matte channel the image. This extra channel usually defines a mask which represents a sort of a cookie-cutter for the image. This is the case when alpha is 255 (full coverage) for pixels inside the shape, zero outside, and between zero and 255 on the boundary. If Afile or Bfile is just a single -, then *comp* reads that file from the standard input.

The operations (specified with -o ) behave as follows (assuming the operation is "A op B"):

- over**    The result will be the union of the two image shapes, with A obscuring B in the region of overlap. This is the default if no operation is specified.
- in**      The result is simply the image A cut by the shape of B. None of the image data of B will be in the result.
- atop**    The result is the same shape as image B, with A obscuring B where the image shapes overlap. Note this differs from **over** because the portion of A outside B's shape does not appear in the result.
- out**     The result image is image A with the shape of B cut out.
- xor**     The result is the image data from both images that is outside the overlap region. The overlap region will be blank.
- plus**    The result is just the sum of the image data. Output values are clipped to 255 (no overflow). This operation is actually independent of the alpha channels.
- minus**   The result of A - B, with underflow clipped to zero. The alpha channel is ignored (set to 255, full coverage).
- diff**    The result of A - B, with underflow wrapping around. This is useful for comparing two very similar images.

## SEE ALSO

*RLE*(5).

"Compositing Digital Images", Porter and Duff, *Proceedings of SIGGRAPH '84* p.255

## AUTHORS

Rod Bogart and John W. Peterson

## BUGS

The other operations could be optimized as much as **over** is.

Comp assumes both input files have the same number of channels.

**NAME**

**crop** – Change the size of an RLE image

**SYNOPSIS**

**crop** *xmin ymin xmax ymax* [ *infile* ] > *outfile*

**DESCRIPTION**

Crop changes the size of an RLE image. The command line numbers *xmin ymin xmax ymax* specify the size of the resulting image. If the resulting image is larger than the original, *crop* supplies blank pixels, otherwise pixels are thrown away.

**SEE ALSO**

*RLE(5)*, *comp(1)*

**AUTHOR**

Rod Bogart

**BUGS**

Could be combined with *repos*. Does not check to see if the input and output regions are disjoint.

## NAME

*dvirle* - convert dvi version 2 files, produced by TeX82, to RLE images

## SYNOPSIS

*dvipr* [ *-m number* ] [ *-h* ] [ *-s* ] [ *-d number* ] [ *-x xfilter* ] [ *-y yfilter* ] *.dvi file name*

## DESCRIPTION

*dvirle* converts .dvi files produced by TeX(1) to RLE(5) format. The basic process involves two passes. In the first pass, the .dvi file is converted into a list of characters. The second pass takes this list and converts it to RLE. The image is filtered to produce gray-scale letters. 300dpi fonts are used, producing an unfiltered page size of approximately 2500x3000 pixels. The default is to average this by 5 pixels in the X direction and 4 in the Y, producing a 510x825 image with an aspect ratio suitable for a Grinnell frame buffer. The filtering parameters can be altered with the *-x* and *-y* flags.

The *-m number* option is used to change the device magnification (which is in addition to any magnification defined in the TeX source file). *number* should be replaced by an integer which is 1000 times the magnification you want. for example, *-m 1315* would produce output magnified to 131.5% of true size. Using device magnification is most useful when you are trying to produce a output for publication. (Many photocopiers will reduce output to 77% of size. When you reduce output printed at 131.5% to 77%, you get close to the true size.) The default is no magnification (1000). Note, however, that a site will only support particular magnifications. If you get error messages indicating that fonts are missing when using this option, you probably have picked an unsupported magnification.

The *-h* flag, when supplied, causes the image to be converted "on its side" (rotated by 90 degrees).

Normally the first pass prints the page numbers from the .dvi file. The *-s* flag suppresses these.

The default *maxdrift* parameter is 2 pixels (1/100th of an inch); the *-d* option may be used to alter this. The *maxdrift* parameter determines just how much font spacing is allowed to influence character positioning. The value 2 allows a small amount of variation within words without allowing any letters to become too far out of position; this had been happening in rare cases before the *maxdrift* code was added.

The output file contains a number of separate RLE images concatenated, one for each page in the input. These should be separated with *rlesplit*(1) before other RLE tools can be used. The output images have a single image channel and an identical "alpha" channel. For compositing with a colored background, it will be necessary to use *rleswap*(1) to expand it to 3 color channels.

The shell script *topcrop* will crop off the top 480 lines of the output image (assuming the default filtering parameters), making it suitable for viewing on a frame buffer.

*topcrop* <file.rle> >cropfile.rle

## FILES

*dvirle1* first pass  
*dvirle2* second pass

## SEE ALSO

*dvisselect*(1), *rlesplit*(1), *rleswap*(1), RLE(5)

## AUTHOR

The original (Versatec) version was written by Janet Incerpi of Brown University. Richard Furuta and Carl Binding of the University of Washington modified the programs for DVI version 2 files. Chris Torek of the University of Maryland rewrote both passes in order to make them run at reasonable speeds. Spencer W. Thomas of the University of Utah converted it to produce RLE images as output.

## BUGS



## NAME

*dvi*select – extract pages from DVI files

## SYNOPSIS

*dvi*select [ *-s* ] [ *-i infile* ] [ *-o outfile* ] *list of pages* [ *infile* [ *outfile* ] ]

## DESCRIPTION

*Dvi*select selects pages from a DVI file produced by TeX, creating a new DVI file usable by any of TeX's conversion program (e.g., *iptex*), or even by *dvi*select itself.

A *range* is a string of the form *first:last* where both *first* and *last* are optional numeric strings, with negative numbers indicated by a leading underscore character “\_”. If both *first* and *last* are omitted, the colon may also be omitted, or may be replaced with an asterisk “\*”. A *page range* is a list of ranges separated by periods. A *list of pages* is described by a set of page ranges separated by commas and/or white space.

*Dvi*select actually looks at the ten *count* variables that TeX writes; the first of these (*\count0*) is the page number, with *\count1* through *\count9* having varied uses depending on which macro packages are in use. (Typically *\count1* is a chapter or section number.) A page is included in *dvi*select's output if all its *\count* values are within any one of the ranges listed on the command line. For example, the command “*dvi*select \*.1,35:” might select everything in chapter 1, as well as pages 35 and up. “*dvi*select 10:30” would select pages 10 through 30 (inclusive). “:43” means everything up to and including page 43 (including negative-numbered pages). If a Table of Contents has negative page numbers, “:\_1” will select it. “\*.4 .....1” might mean everything in every chapter 4 and an index, presuming *\count9* was set to 1 in the index. (“\*” must be quoted from the shell; the null string is more convenient to use, if harder to read.)

Instead of *\count* values, *dvi*select can also select by “absolute page number”, where the first page is page 1, the second page 2, and so forth. Absolute page numbers are indicated by a leading equal sign “=”. Ranges of absolute pages are also allowed: “*dvi*select =3:7” will extract the third through seventh pages. Dot separators are not legal in absolute ranges, and there are no negative absolute page numbers.

More precisely, an asterisk or a null string implies no limit; an equal sign means absolute pages rather than *\counts*; a leading colon means everything up to and including the given page; a trailing colon means everything from the given page on; and a period indicates that the next *\count* should be examined. If fewer than 10 ranges are specified, the remaining *\counts* are left unrestricted (that is, “1:5” and “1:5.\*” are equivalent). A single number *n* is treated as if it were the range *n:n*. An arbitrary number of page selectors may be given, separated by commas or whitespace; a page is selected if any of the selectors matches its *\counts* or absolute page number.

*Dvi*select normally prints the page numbers of the pages selected; the *-s* option suppresses this.

## AUTHOR

Chris Torek, University of Maryland

## SEE ALSO

*dvirle*(1), *The TeXbook*

## BUGS

A leading “-” ought to be allowed for negative numbers, but it is currently used as a synonym for “:”, for backwards compatibility.

Section or subsection selection will sometimes fail, for the DVI file lists only the *\count* values that were active when the page ended. Clever macro packages can alleviate this by making use of other “free” *\count* registers. Chapters normally begin on new pages, and do not suffer from this particular problem.

*Dvi*select does not adjust the parameters in the postamble; however, since these values are normally used only to size certain structures in the output conversion programs, and the parameters never need to be adjusted upward, this has not proven to be a problem.

## NAME

*fant* - perform simple spatial transforms on an image

## SYNOPSIS

*fant* [ **-s** *xscale yscale* ] [ **-v** ] [ **-a** *angle* ] [ **-o** *xoff yoff* ] [ *infile* ] *outfile*

## DESCRIPTION

*fant* rotates or scales an image by an arbitrary amount. It does this by using pixel integration (if the image size is reduced) or pixel interpolation if the image size is increased. Because it works with subpixel precision, aliasing artifacts are not introduced. *Fant* uses a two-pass sampling technique to perform the transformation.

The following options are available:

**-s xscale yscale**

The amount (in real numbers) to scale an image by. This is often useful for correcting the aspect of an image for display on a frame buffer with non square pixels. For this use, the origin should be specified as 0, 0 (see below). If an image is only scaled in Y and no rotation is performed, *fant* only uses one sampling pass over the image, cutting the computation time in half.

**-a angle**

Amount to rotate image by, a real number from 0 to 45 degrees (positive numbers rotate clockwise). Use *rleflip(1)* first to rotate an image by larger amounts.

**-o xoff yoff** Specifies where the origin of the image is - the image is rotated or scaled about this point. If no origin is specified, the center of the image is used.

## SEE ALSO

*avg4(1)*, *rleflip(1)* *RLE(5)*.

## AUTHOR

John W. Peterson

## BUGS

*fant* was implemented with floating point arithmetic. It would undoubtedly run faster if integer arithmetic was used.

Negative rotations are implemented but don't quite work right.

**NAME**

*flip\_book* – view a sequence of frames in real time

**SYNOPSIS**

*flip\_book* files...

**DESCRIPTION**

*flip\_book* generates a real-time sequence of images on some Apollo color displays. It loads the images into the frame buffer zooms the frame buffer so one frame fills the screen, then BLT's the frames into the visible portion of the screen in rapid succession.

*flip\_book* takes as arguments a number of files to display, in the sequence they should appear. The files should be eight bit images (either black and white or dithered), and have a uniform power of two size (e.g., 512x512, 256x256, 128x128, etc). While *flip\_book* is running, the following keys are active:

*space* Step through the image one frame at a time.

*f* Run at film speed (24 frames per second).

*v* Run at video speed (30 frames per second).

*F* Run fast (60 frames per second).

*b* Back and forth mode.

**SEE ALSO**

*getap(1)* *RLE(5)*.

**AUTHOR**

John W. Peterson

**BUGS**

Runs only on DN550 and DN660 style displays.

Somebody should write a version for X.

## NAME

getap - get RLE images to an Apollo display

## SYNOPSIS

getap [ -a ] [ -l ] [ -g gamma ] [ -b ] [ -i ] [ file ]

## DESCRIPTION

This program displays an *RLE(5)* file on an Apollo workstations running the Display Manager. It uses a dithering technique to take a full-color or gray scale image into the limited number of colors typically available, unless "borrow mode" is specified. Under borrow mode, the 24 bit mode of the Apollo hardware is used (if it's available). On bitmap displays, *getap* converts the image to black and white and uses bitmap dithering.

The following options are available:

-a      Uses the Alpha\_1 "perceptual" color map.

-l      Loads a linear color map.

-g gamma

        This load a color map with the specified gamma.

-b      This tells *getap* to use "borrow mode" instead of an apollo window, if the hardware supports it. The only hardware that supports this are the DN550, DN560 and the DN660. The bottom portion of the image is chopped off on workstations without square screens.

-i      On black and white displays, this flips the orientation of black and white.

## SEE ALSO

*GETX(1)*, *RLE(5)*.

## AUTHOR

John W. Peterson

## BUGS

*Getap* is pretty sloppy about dealing with the color map, particularly in window mode.

Since Apollo workstations now support the X window system, *getap* is mostly subsumed by *getX*.

## NAME

`getOrion` – get RLE images to an Orion graphics display

## SYNOPSIS

`getOrion [ -D ] [ -a ] [ -b ] [ -f ] [ -g gam ] [ -l ] [ -r ] [ file ]`

## DESCRIPTION

This program displays an *RLE(5)* file on a High Level Hardware Orion graphics display running the Star-Point graphics system. It uses a dithering technique to take a full-colour or grey scale image into the limited number of colours available.

The default behaviour is to display the image in colour using a 216 colour map (6 intensities per primary). However, an *RLE(5)* file with 1 colour and 3 colour map channels is treated as a special case with the colour map in the header loaded as the graphics colour map and the data used to index this map. In this mode of operation no dithering is done as the file is assumed to be the output of some program which has selected the "best" possible colours for the image and has already corrected some of the errors produced by the quantisation. An option is provided to force a grey scale display of colour images.

`getOrion` uses the standard window manager creation procedure to create a window at a particular location on the screen. The size of the window is the size of the image.

The following options are available:

- `-D` "Debug mode". The operations in the input *RLE(5)* file will be printed as they are read.
- `-a` Uses the Alpha1 perceptual correction table for gamma correcting the colour map.
- `-b` Forces `getOrion` to produce a grey scale dithered image instead of a colour image using 128 shades of grey. Colour input will be transformed to grey level using the NTSC Y transform.
- `-f` Normally `getOrion` will only use entries 0-239 of the graphics device colour map, as the others are used by the window manager for background, icons, etc. This option will force it to use all 256 entries and is useful only when the image has been specified with a 24-bit colour map
- `-g gam` Specifies, as a floating point number, the gamma correction factor to be used when correcting the colour map.
- `-l` Use a linear colour map. Identical to having a gamma of 1.
- `-r` Use "reverse" mode for display. The scanlines are by default displayed from the bottom-up, this option displays them from the top-down. Useful for applications which have produced the scanlines starting from the top one.
- `file` Name of file to display. If none specified, the image will be read from standard input.

## SEE ALSO

*RLE(5)*, *getX(1)*.

## AUTHOR

Gianpaolo Tommasi, Computer Laboratory, University of Cambridge. The code is based on other "get" routines.

**DEFICIENCIES**

The window cannot be moved whilst the image is being displayed.

Because of the way the graphics memory is organised displaying images in GM\_BW mode is slow.

## NAME

getX – get RLE images to an X display

## SYNOPSIS

```
getX [ -{bB} ] [ -m ] [ -f ] [ -p ] [ -z ] [ -{cwW} ] [ -D ] [ -l levels ] [ -d display ] [ ==
window_geometry ] [ -{iI} image_gamma ] [ -g display_gamma ] [ file ]
```

## DESCRIPTION

This program displays an *RLE(5)* file on an X display. It uses a dithering technique to take a full-color or gray scale image into the limited number of colors typically available under X. Its default behavior is to try to display the image in color with as many brightness levels as possible (except on a one bit deep display), options are provided to limit the number of levels or to force black and white display. Several *getX* processes running simultaneously with the same color resolution will share color map entries.

Other options allow control over the gamma, or contrast, of the image. The dithering process assumes that the incoming image has a gamma of 1.0 (i.e., a 200 in the input represents an intensity twice that of a 100.) If this is not the case, the input values must be adjusted before dithering via the *-i* or *-I* option. The input file may also specify the gamma of the image via a picture comment (see below). The output display is assumed to have a gamma of 2.5 (standard for color TV monitors). This may be modified via the *-g* option if a display with a different gamma is used.

*getX* uses the standard X window creation procedure to create a window with a location and size specified by the user, with the restriction that the window must be at least as large as the input image. If the window is turned into an icon, a smaller version of the image will be displayed in the icon. A shifted mouse click on either the window or icon will cause the image to be removed.

The following options are available:

- b After displaying the image in a window, *getX* will attempt to set your "root" window background tiling pattern to the image. There are some strict limitations on image size for this to work (at least in X10). A color or gray-scale image must be smaller than 256x256, and a black and white (-W) image smaller than about 720x720. If the image is larger than this, a strip from the top of the image will be displayed in the background. Note that if you kill the *getX* window, the color map entries will not be protected; any other program that asks for a color map entry will likely get one that is being used by the background image.
- B Loads the image into the background as above, but does not display it in a window. *getX* exits after loading the background, leaving the color map unprotected, as above.
- m Just loads the color map. This may be suitable for fixing up the color map used by the root background.
- f Normally, *getX* will fork itself after putting the image on the screen, so that the parent process may return the shell, leaving an "invisible" child to keep the image refreshed. If *-f* is specified, *getX* will not exit to the shell until the image is removed.
- z This flag creates a zoom window for the image. The new window is created by the standard X window creation process. The mouse can be used in the image window to select the area to zoom. Pressing any button will reset the center of the zoom window to be the selected pixel. A clickdrag in the image window will resize the zoom window to enclose the selected region. Pressing the left button in the zoom window will decrease the zoom factor, but will keep the same number of pixels zoomed. The right button increases the zoom factor. If the middle button is pressed in the zoom window, position information will be printed for the selected zoom pixel. Note that the info

will be printed only if `-f` is given with the `-z` option. One may also resize the zoom window to change the number of pixels that are zoomed.

- `-p` `getX` tries to copy the image to an off-screen pixmap for quick refresh. On some displays, this will fail if no off-screen memory is available. The image will disappear shortly after it is completed when this happens. You should specify `-p` to prevent the attempt to use a pixmap.
- `-w` This flag forces `getX` to produce a gray scale dithered image instead of a color image. Color input will be transformed to black and white via the *NTSC Y* transform. On a low color resolution display (a display with only 4 bits, for example), this will produce a much smoother looking image than color dithering. It may be used in conjunction with `-l` to produce an image with a specified number of gray levels.
- `-W` This flag forces `getX` to display the image as a black and white bitmap image. This is the only mode available on monochrome (non gray scale) displays (and is the default there). Black pixels will be displayed with pixel value 0 and white with pixel value 1 (note that these may not be black and white on certain displays, or if they have been modified with *xset*.)
- `-c` This flag suppresses all dithering, and causes `getX` to load the color map in the image file directly into the display. Channel 0 of the image will be treated as a set of indices into the color map. If there are not enough color map entries available in the display, as many as fit will be loaded and all other "colors" will be mapped to black. The picture comment *color\_map\_length=maplen* can be used to specify the exact number of relevant color map entries.
- `-D` "Debug mode". The operations in the input *RLE(5)* file will be printed as they are read.
- `-l levels`  
Specify the number of gray or color levels to be used in the dithering process. If not this many levels are available, `getX` will try successively fewer levels until it is able to allocate enough color map entries.
- `-d display`  
Give the name of the X display to display the image on. Defaults to the value of the environment variable *DISPLAY*.
- `-- window_geometry`  
Specify the geometry of the window in which the image will be displayed. This is useful mostly for giving the location of the window, as the size of the window will be at least as large as the size of the image. The *window\_geometry* specification need not begin with an "=" sign.
- `-i image_gamma`  
Specify the gamma (contrast) of the image. A low contrast image, suited for direct display without compensation on a high contrast monitor (as most monitors are) will have a gamma of less than one. The default image gamma is 1.0. Image gamma may also be specified by a picture comment in the *RLE(5)* file of the form *image\_gamma=gamma*. The command line argument will override the value in the file if specified.
- `-I image_gamma`  
An alternate method of specifying the image gamma, the number following `-I` is the gamma of the display for which the image was originally computed (and is therefore 1.0 divided by the actual



gamma of the image). Image display gamma may also be specified by a picture comment in the *RLE(5)* file of the form `display_gamma=gamma`. The command line argument will override the value in the file if specified.

**-g display\_gamma**

Specify the gamma of the X display monitor. The default value is 2.5, suitable for most color TV monitors (this is the gamma value assumed by the NTSC video standard).

**file**      Name of the *RLE(5)* file to display. If not specified, the image will be read from the standard input.

**SEE ALSO**

*RLE(5)*.

**AUTHOR**

Spencer W. Thomas, University of Utah

**BUGS**

It gets an X error when displaying an image only one line high.

**DEFICIENCIES**

It totally ignores the *Xdefaults* file.

## NAME

`getcx3d` – display an RLE(5) image on the Chromatics

## SYNOPSIS

`getcx3d [ -O ] [ -B ] [ -d ] [ -t ] [ -p x y ] [ -l ] [ file ] ...`

## DESCRIPTION

This program displays an *RLE(5)* image on a Chromatics CX 1536 (raster dimensions 1536×1152×24) running CX3D. If GKS is booted on the CX, then use the programs *getcx(1)* and *getcxr(1)*.

*Getcx3d* will display black and white and full color images, ignoring the alpha channel if present. All three background styles of the *RLE(5)* format are supported: (0) write every pixel, (1) do not write background pixels (overlay) and (2) clear to background; see the `-O` and `-B` options. You may position an image at some place other than (0, 0) on the screen; see the `-p` option. The `-d` and `-t` options magnify the image; see below. The bounding box of the image is the only part of the image that is ever displayed (i.e. clear to background will only clear the area within the bounding box, not the entire screen.) The color maps within the CX are not changed. Colors are passed through a gamma correction map ( $\text{gamma\_value} = 2.5$  in  $\text{round}(255 * ((x / 255) ^ (1 / \text{gamma\_value})))$ , judged best for the monitor connected to the CX) on the host before they are sent to the CX. Use `-l` to pass colors through a linear map. Finally, any color maps specified by the RLE file are ignored. This is a bug.

The following options are available:

- `-O` Force overlay background style. Ignoring the background style indicated in the RLE file this option will overlay the RLE image, causing the previous image on the CX to show through pixels of background color of the present image.
- `-B` Force clear to background style. Ignoring the background style indicated in the RLE file this option will clear the bounding box area of the RLE file before displaying the image.
- `-d` Double the image size. Display four pixels for every one pixel of the RLE file.
- `-t` Triple the image size. Display nine pixels for every one pixel of the RLE file.
- `-p x y` Reposition the image. Place the left corner of the image (0, 0) at some place other than the left corner of the CX. Note that the left corner of the image is (0, 0), which may be different from the left corner of the bounding box of the image. The bounding box is the only area of the image that is ever displayed.
- `-l` Use a linear map. By default all colors are passed through a gamma correction map ( $\text{gamma\_value} = 2.5$  in  $\text{round}(255 * ((x / 255) ^ (1 / \text{gamma\_value})))$ , judged best for the monitor connected to the CX) on the host before they are sent to the CX. This option causes no mapping to take place.
- `file` Name of file to restore. If not specified or if `-`, an RLE encoded image is read from the standard input. If the specified file is not found, then the file is searched for in `/w/tmp/images` and `/w/tmp/wtm` in that order.

Any number of images may be displayed with one invocation of *getcx3d*. Options, however, are restored to their default values immediately after the display of each file listed on the command line.

Other options to come.

**FILES**

/dev/dr0

**SEE ALSO**

*getc*(1), *getc**rx*(1), *svfb*(1), *getfb*(1), *cx3dzoom*(1), *cx3dclr*(1), *RLE*(5).

**SOURCE**

/usr/site/src/graphics/cx/getcx3d

**AUTHOR**

W. Thomas McCollough, Jr., University of Utah

**BUGS**

Color maps are not loaded.

If interrupted with a catchable signal, *getc**x3d* will close the CX gently, allowing future access without rebooting. If *getc**x3d* is stopped, however, and then (before it is continued) killed with any signal, then the CX may be left in a bad state.

**NAME**

getfb - display an RLE file on a BRL libfb frame buffer.

**SYNOPSIS**

getfb [ -d ] [ rlefile ]

**DESCRIPTION**

This program displays an *RLE(5)* file on any frame buffer supported by the BRL *libfb*. The option *-d* prints the image header information and turns on debugging of the *RLE* file. All of the *RLE* opcodes will be printed as they are read from the input file. If an input *rlefile* is not specified, input will be taken from standard input.

**SEE ALSO**

*RLE(5)*.

**AUTHOR**

Paul Stay, Ballistic Research Laboratory.

**LIMITATIONS**

Will only display images up to 1024 pixels wide. Will not display black and white (single channel) images correctly. Ignores color map in *RLE* file.

**NAME**

getgmr – Restore an RLE image to a Grinnell GMR-27 frame buffer.

**SYNOPSIS**

getgmr [ -q ] [ -d ] [ -BO ] [ -pi x y ] [ -c channel [ into ] ] [ file ]

**DESCRIPTION**

Displays an *RLE(5)* file on a Grinnell GMR-27 frame buffer.

- q      Query the given file. Determine if it is an *RLE(5)* file. Does not affect the frame buffer.
- d      Debug the given file. Print information about each command in the input file. Displays as it prints.
- B      If the file was saved with -B or -O, restore the background color before restoring the image data.
- O      If the file was saved with -B or -O, restore the image data in overlay mode. Only areas of the original image which were not the background color are restored. The rest of the image already in the frame buffer is undisturbed.
- p x y    Reposition the image. The original lower left corner is positioned at [x, y] before restoring the image. A warning: A saved image should not be repositioned so that any saved data wraps around the X borders. If the file was not saved with -B or -O, this includes background areas.
- i x y    Reposition the image incrementally, that is, x and y are taken as offsets from the original position of the image.
- c channel [ into ]  
Put only the given color *channel* into the frame buffer. If *into* is specified, loads it into that *channel*. If the input file is black and white (one channel), then -c *channel* is equivalent to -c 0 *channel*.
- file      Name of file to restore. If not specified, input is read from stdin.

**SEE ALSO**

*RLE(5)*.

**AUTHOR**

Spencer W. Thomas  
Todd Fuqua

**BUGS**

Seems to interact poorly with Grinnell hardware bugs at times.

**NAME**

getiris - display an RLE image on a Silicon Graphics Iris Workstation.

**SYNOPSIS**

getiris [ file ]

**DESCRIPTION**

This program displays an *RLE(5)* file on a *Silicon Graphics Iris* that is not running the *mex* window manager. It uses the full 24 bits of color available on an iris. After the picture is displayed, press any mouse button to erase the screen.

The following options are available:

file     Name of the *RLE(5)* file to display. If not specified, the image will be read from the standard input.

**SEE ALSO**

*RLE(5)*, *GETMEX(1)*.

**AUTHOR**

Glenn McMinn and Rod Bogart, University of Utah.

## NAME

getmex - get RLE images to an X display

## SYNOPSIS

getmex [ -f ] [ -w ] [ -D ] [ -m mapstart ] [ file ]

## DESCRIPTION

This program displays an *RLE(5)* file on a *Silicon Graphics Iris* display running the *mex* window manager. It uses a dithering technique to take a full-color or gray scale image into the limited number of colors typically available under *mex*. Its default behavior is to try to display the image in color, an option is provided to force black and white display. Several *getmex* processes running simultaneously will share color map entries.

*getmex* uses the standard *mex* window creation procedure to create a window with a location and size specified by the user, with the restriction that the window will be no larger than the input image. If the window is smaller than the image, the center of the image will be visible in the window.

You can "pan" a small window around in an image by attaching the mouse to the window using *mex* and clicking the *left mouse button* in the image. The position in the image under the cursor will jump to the center of the window. The *setup* key resets the view to position the center of the image in the center of the window. Furthermore, *control-setup* saves the current view, and *shift-setup* restores it.

The following options are available:

- f Normally, *getmex* will fork itself after putting the image on the screen, so that the parent process may return the shell, leaving an "invisible" child to keep the image refreshed. If -f is specified, *getmex* will remain attached to the shell, whence it may be killed with an interrupt signal or via *mex*.
- w This flag forces *getmex* to produce a gray scale dithered image instead of a color image. Color input will be transformed to black and white via the *NTSC Y* transform. Since a 128-step greyscale is used, this will produce a much smoother looking image than color dithering.
- D "Debug mode". The operations in the input *RLE(5)* file will be printed as they are read.
- m mapstart  
Specifies the starting location of the block of color map to be used by *getmex*. (There are 1024 colors available on the current *Irises* under *mex*.) The default for color images is a block of 512 rgb colors starting at location 512 in the color map. Black-and-white images default to a block of 128 grey shades starting at location 128. Both the rgb and grey ramps are gamma-corrected in the same way as the *makemap* program in the */usr/gifs/mextools/tools* directory. You probably want to set up your initial color map using *makemap*.
- file Name of the *RLE(5)* file to display. If not specified, the image will be read from the standard input.

## SEE ALSO

*RLE(5)*.

## AUTHOR

Russ Fish, University of Utah. Based on *getX*, by Spencer W. Thomas.





**NAME**

getren - get RLE images to an HP98721 ("Renaissance") display

**SYNOPSIS**

getren [ **-p** *xpos ypos* ] [ **-O** ] [ **-i** *xoff yoff* ] [ *file* ]

**DESCRIPTION**

This program displays an *RLE(5)* file on an HP 98721 "Renaissance" display configured with at least 24 bits per pixel. If a color map exists in the file, it is loaded into the display, otherwise a linear map is used.

The following options are available:

**-p** *xpos ypos*

position the image at *xpos*, *ypos*.

**-O** Don't clear the screen (overlay mode)

**-i** *xoff yoff*

Offset the image position by *xoff yoff*

**SEE ALSO**

*GETX(1)*, *RLE(5)*.

**AUTHOR**

John W. Peterson, University of Utah, with input from Filippo Tampieri of Cornell and Eric Haines of 3D/Eye.

**BUGS**

The program assumes a full 24 bit Renaissance display. The HP graphics library supports automatic dithering for displays with fewer bitplanes, but getren ignores this.

The device and driver names are compiled in as *"/dev/crtren"* and *"hp98721"*, respectively. This may need changing on systems configured differently (in particular, systems with the Renaissance as their sole display may use a different name for the device).

## NAME

`getsun` - get RLE images to a sun window

## SYNOPSIS

`getsun [ -{wW} ] [ -D ] [ -l levels ] [ -{iI} image_gamma ] [ -g display_gamma ] [ file ]`

## DESCRIPTION

This program displays an *RLE(5)* file in a sun window display. It uses a dithering technique to take a full-color or gray scale image into the limited number of colors available under sun windows. Its default behavior is to try to display the image in color with as many brightness levels as possible (except on a one bit deep display), options are provided to limit the number of levels or to force black and white display. Several *getsun* processes running simultaneously with the same color resolution will share color map entries.

Other options allow control over the gamma, or contrast, of the image. The dithering process assumes that the incoming image has a gamma of 1.0 (i.e., a 200 in the input represents an intensity twice that of a 100.) If this is not the case, the input values must be adjusted before dithering via the `-i` or `-I` option. The input file may also specify the gamma of the image via a picture comment (see below). The output display is assumed to have a gamma of 2.5 (standard for color TV monitors). This may be modified via the `-g` option if a display with a different gamma is used.

*getsun* creates a sun window the size of the image being displayed. The header of the new window displays the name of the image being displayed and its size. The following options are available:

- `-w` This flag forces *getsun* to produce a gray scale dithered image instead of a color image. Color input will be transformed to black and white via the *NTSC Y* transform. On a low color resolution display (a display with only 4 bits, for example), this will produce a much smoother looking image than color dithering. It may be used in conjunction with `-l` to produce an image with a specified number of gray levels.
- `-W` This flag forces *getsun* to display the image as a black and white bitmap image. This is the only mode available on monochrome (non gray scale) displays (and is the default there). Black pixels will be displayed with pixel value 0 and white with pixel value 1.
- `-D` "Debug mode". The operations in the input *RLE(5)* file will be printed as they are read.
- `-l levels`  
Specify the number of gray or color levels to be used in the dithering process. The default is 5 except on monochrome (non gray scale) displays. Levels must be in the range [2,6].
- `-i image_gamma`  
Specify the gamma (contrast) of the image. A low contrast image, suited for direct display without compensation on a high contrast monitor (as most monitors are) will have a gamma of less than one. The default image gamma is 1.0. Image gamma may also be specified by a picture comment in the *RLE(5)* file of the form `image_gamma=gamma`. The command line argument will override the value in the file if specified.
- `-I image_gamma`  
An alternate method of specifying the image gamma, the number following `-I` is the gamma of the display for which the image was originally computed (and is therefore 1.0 divided by the actual gamma of the image). Image display gamma may also be specified by a picture comment in the *RLE(5)* file of the form `display_gamma=gamma`. The command line argument will override the

value in the file if specified.

**-g display\_gamma**

Specify the gamma of the *sun* display monitor. The default value is 2.5, suitable for most color TV monitors (this is the gamma value assumed by the NTSC video standard).

**file**      Name of the *RLE(5)* file to display. If not specified, the image will be read from the standard input.

**SEE ALSO**

*RLE(5)*.

**NAME**

into - copy into a file without destroying it

**SYNOPSIS**

into [ -f ] file

**DESCRIPTION**

*Into* copies its standard input into the specified *file*, but doesn't actually modify the file until it gets EOF. This is useful in a pipeline for putting stuff back in the "same place." The *file* is not overwritten if that would make it zero length, unless the -f option is given. That option also forces overwriting of the *file* even if it is not directly writable (as long as the directory is writable).

**SEE ALSO**

pipe(2)

**BUGS**

For efficiency reasons, the directory containing the *file* must be writable by the invoker. The original *file*'s owner is not preserved.

**NAME**

**mcut** – Quantize colors in an image using the median cut algorithm

**SYNOPSIS**

**mcut** [ **-d colors** ] file > output

**DESCRIPTION**

*Mcut* reads an RLE file and tries to choose the "best" subset of colors to represent the colors present in the original image. A common use for this is to display a 24 bit image on a frame buffer with only eight bits per pixel using a 24 bit color map. *Mcut* first quantizes intensity values from eight bits to five bits, and then chooses the colors from this space.

*Mcut* runs in two passes; the first pass scans the image to find the color distributions, and the second pass maps the original colors into color map indices. The output file has a color map containing the colors *mcut* has chosen. *Mcut* also sets the picture comment "color\_map\_length" equal to the number of colors it has chosen. The *getX* program can be told to use this color map instead of dithering with the **-c** flag.

The following option is available:

**-d ncolors**

Limit the number of colors chosen to *ncolors*. The default is 200.

**SEE ALSO**

*GETX*(1),

"Color Image Quantization for Frame Buffer Display", by Paul Heckbert, Proceedings of SIGGRAPH '82, July 1982, p. 297.

**AUTHOR**

Robert Mecklenburg, John W. Peterson, University of Utah.

**BUGS**

*Mcut* often produces visible contours on smooth shaded images. The output should be dithered to avoid this artifact.

The initial quantization is hardwired to five bits. This should be a compile-time option.

**NAME**

`mergechan` – merge channels from several RLE files into a single output stream

**SYNOPSIS**

`mergechan [ -a ] files... > output`

**DESCRIPTION**

*Mergechan* takes input from several RLE files and combines them into a single output stream. Each channel in the output stream comes from the respective filename specified on the input (i.e., channel zero is taken from the first file, channel one from the next, etc). The same file can be specified more than once. If the `-a` flag is given, the channels are numbered from -1 (the alpha channel) instead of zero. All of the input channels must have exactly the same dimensions (use *crop*(1) to adjust files to fit each other).

*Mergechan* is typically used to introduce an alpha mask from another source into an image, or combine color channels digitized independently.

**SEE ALSO**

*RLESWAP*(1), *urt/etc/mix.c*

**AUTHOR**

John W. Peterson, University of Utah.

**BUGS**

*Mergechan* is totally ignorant of the color maps of the input files.

The restriction that all input files must be the same size could probably be removed.

It's not well tested.

**NAME**

**painttorle** – Convert MacPaint images to RLE format.

**SYNOPSIS**

**painttorle** [ **-i** ] [ *red green blue alpha* ] *infile.paint* | **rleflip -v** > *outfile.rle*

**DESCRIPTION**

*Painttorle* converts a file from MacPaint format into RLE format. Because MacPaint and RLE disagree on which end is up, the extra *rleflip* is used to preserve orientation. If no input file is specified, the data is read from stdin. The following options are available:

*red green blue alpha*

Allows the color values to be specified (the default is 255).

**-i** Invert the color of the MacPaint pixels.

**AUTHOR**

John W. Peterson

## NAME

**pyrmask** – Blend two images together using Gaussian pyramids.

## SYNOPSIS

**pyrmask inmask outmask maskfile [ -o outfile ]**

## DESCRIPTION

*Pyrmask* blends two images together by first breaking the images down into separate bandpass images, combining these separate images, and then adding the new bandpass images back into a single output image. This can produce very seamless blends of digital images. The two images are combined on the basis of a third "mask" image. The resulting image will contain the *inmask* image where the mask contains a maximum value (255) and the *outmask* image where the mask contains zeros. This is done on a channel by channel basis, i.e. the maskfile should have data in each channel describing how to combine each channel of the inmask and outmask images. All three images must have exactly the same dimensions (both image size and number of channels). For best results, it's often useful to filter the mask image a little with *smush*(1) first.

## SEE ALSO

*SMUSH*(1), *RLESWAP*(1)

Burt and Adelson, "A Multiresolution Spline With Applications to Image Mosaics", ACM Transactions on Graphics, October 1983, V2 #4, p. 217.

Ogden, Adelson, Bergen and Burt, "Pyramid-based Computer Graphics", RCA Engineer, Sept/Oct 1985, p. 4.

## AUTHOR

Rod Bogart

## BUGS

Pyrmask was included in the distribution as a last minute decision, it still has several rough edges. The current implementation has very strict requirements for image sizes and dimensions. The extensive use of floating point computation makes it very slow for normal sized images. It also keeps all of the bandpass images in core at once, which requires considerable amounts of memory.

Pyrmask is build on top of a library of functions for working with Gaussian pyramids. This library has yet to be documented or fully tested.



**NAME**

read98721 – read an image from the HP-98721 frame buffer

**SYNOPSIS**

read98721 [ -O ] [ -m ] [ -p xpos ypos ] [ -s xsize ysize ] [ -d display ] [ -x driver ] [ -b rbak gbak bbak ] [ -o file ] [ comments ... ]

**DESCRIPTION**

This program reads an image from a *HP-98721* frame buffer and writes it to an *RLE(5)* file. The file will contain three channels of 8 bits each for red, green, and blue respectively. If an output file name is not specified the image will be written to the standard output. The default display device and device driver are respectively */dev/crt98721* and *hp98721*.

The following options are available:

- O      Specifies that the image has no background.
- m      Saves the device color maps. By default, no color maps are saved.
- p xpos ypos  
      Specifies the lower left corner of the portion of the screen to be saved. The origin is the lower left corner of the display, which is taken as the default starting position if this option is not specified.
- s xsize ysize  
      Specifies the size of the image to be read.
- d display  
      Gives the name of the display device from which the image is to be read.
- x driver  
      Gives the name of the device driver to be used to communicate with the display device.
- b rbak gbak bbak  
      Specifies red, green and blue pixel values for the background.
- o file    Writes the image to *file*.

The remaining arguments are taken to be comment strings of the form *name=value*, and are inserted in the header of the *RLE(5)* output file.

**SEE ALSO**

*RLE(5)*, *get98721(1)*.

**AUTHOR**

Filippo Tampieri, Program of Computer Graphics, Cornell University.

## NAME

*repos* – reposition an RLE image

## SYNOPSIS

*repos* [ *-p xpos ypos* ] [ *-i xinc yinc* ] [ *infile* ] > *outfile*

## DESCRIPTION

*repos* repositions an RLE image. *Repos* just changes the coordinates stored in the RLE header (see *RLE(5)*), no modification is made to the image itself. If no input file name is given, the image is read from standard input.

If neither of the following flags are specified, *-p 0 0* is assumed.

*-p xpos ypos*

Reposition the image to the absolute coordinates *xpos ypos*.

*-i xinc yinc*

Move the image by *xinc yinc* pixels from where it currently is (relative movement).

## DIAGNOSTICS

*Repos* does not allow the image origin to have negative coordinates.

## SEE ALSO

*rlesetbg(1)*. *RLE(5)*.

## AUTHORS

Rod Bogart, John W. Peterson

## BUGS

**NAME**

**rleaddcom** – add picture comments to an RLE file.

**SYNOPSIS**

**rleaddcom** [ **-d** ] [ **-f** *rlefile* ] *comments* ...

**DESCRIPTION**

The *rleaddcom* program will add one or more comments to an *RLE(5)* file. If the option **-f** *rlefile* is not given, it will read from the standard input. The modified *RLE(5)* file is written to the standard output. All remaining arguments on the command line are taken as comments. Comments are nominally of the form *name=value* or *name*. Any comment already in the file with the same *name* will be replaced.

The option **-d** will cause matching comments to be deleted, no comments will be added in this case.

**AUTHOR**

Spencer W. Thomas, University of Utah

**SEE ALSO**

*RLE(5)*, *rlehdr(1)*.

## NAME

*rlebg* - generate simple backgrounds

## SYNOPSIS

*rlebg red green blue [ alpha ] [ -l ] [ -v [ top [ bottom ] ] ] [ -s xsize ysize ] [ outfile ]*

## DESCRIPTION

*rlebg* generates a simple background. These are typically used for compositing below other images. The values *red green blue* specify the pixel values (between 0 and 255) the background will have. If *alpha* is not specified, it defaults to 255 (full coverage) *rlebg* generates both constant backgrounds and backgrounds with continuous ramps. If no output filename is given, the background is written to standard output.

The following options are available:

**-s xsize ysize**

This is the size of the background image. The default is 512x480.

**-l**

Generate a linear ramp of pixel values. If no ramp flag is given, *rlebg* generates a constant background.

**-v top bottom**

Generate a variable ramp, using a square function (this looks best with gamma corrected images). *top* and *bottom* are the fractions of the full color values at the top and bottom of the image. The defaults are 1.0 0.1, respectively. If both **-v** and **-l** are given, then a linear ramp function is used instead of a squared ramp.

## SEE ALSO

*rresetbg*(1). *RLE*(5).

## AUTHOR

Rod Bogart

## BUGS

Because of the way *scanargs*(3) works, the color values must be specified before any of the options.

**NAME**

**rlebox** – print bounding box for image in an RLE file.

**SYNOPSIS**

**rlebox** [ **-v** ] [ **-c** ] [ *rlefile* ]

**DESCRIPTION**

This program prints the bounding box for the image portion of an *RLE(5)* file. This is distinct from the bounds in the file header, since it is computed solely on the basis of the actual image. All background pixels are ignored.

The following options are available:

- v**      Verbose mode: label the numbers for human consumption.
- c**      Print the numbers in the order that *crop* wants them on its command line. The default order is *xmin xmax ymin ymax*. If this option is specified, the bounds are printed in the order *xmin ymin xmax ymax*. Thus, a file *foo.rle* could be trimmed to the smallest possible image by the command  
          *crop 'rlebox -c foo.rle' foo.rle*

*rlefile*    Name of the *RLE* file.

**SEE ALSO**

*crop(1)*, *RLE(5)*.

**AUTHOR**

Spencer W. Thomas, University of Utah

**NAME**

**rleflip** – Invert, reflect or rotate an image.

**SYNOPSIS**

**rleflip** { **-r l h v** } [ infile ] > outfile

**DESCRIPTION**

*Rleflip* inverts, reflects an image; or rotates left or right by 90 degrees. The pictures origin remains the same. If no input file is specified, the image is read from standard input.

One of the following flags must be given:

- r**     Rotate the image 90 degrees to the right
- l**     Rotate the image 90 degrees to the left
- h**     Reflect the image horizontally
- v**     Flip the image vertically

**SEE ALSO**

*fant*(1). *RLE*(5).

**AUTHOR**

John W. Peterson

**NAME**

**rlehdr** – Prints the header of an RLE file

**SYNOPSIS**

**rlehdr** [ *rlefile* ]

**DESCRIPTION**

This program prints the header of an *RLE(5)* file in a human readable form. If the optional *rlefile* argument is not supplied, input is read from standard input.

**SEE ALSO**

*RLE(5)*.

**AUTHOR**

Spencer W. Thomas, University of Utah

**NAME**

**rlehisto** - Create simple histograms for RLE images.

**SYNOPSIS**

**rlehisto** [ **-r** ratio ] [ **-h** height ] [ **-t** ] file [ **-o** outfile ]

**DESCRIPTION**

*Rlehisto* creates a simple histogram graph of the number of times each pixel value occurs in an rle file. The graph reads with the rightmost bar indicating the number of zero pixels, to the leftmost indicating the number of 255 pixels. The height of a given bar is determined by the number of times that pixel value occurs in the file. Each channel's histogram is computed independently, and the results are graphed in the corresponding channel of the output image from *rlehisto*.

The following option is available:

**-r ratio**

Determines the overall scale of the bars. The larger the number, the shorter the graph.

**-h height**

Specifies the absolute height of the output image (the graph may exceed this height).

**SEE ALSO**

*TOBW*(1), *SWAPCHAN*(1)

**AUTHOR**

Rod Bogart, University of Utah.

**BUGS**

Could be lots smarter about automatically scaling the output image on the basis of the size of the histogram.



## NAME

**rleldmap** – Load a new color map into an RLE file

## SYNOPSIS

**rleldmap** [ *-{a,b}* ] [ *-n nchan length* ] [ *-s bits* ] [ *-l [ factor ]* ] [ *-g gamma* ] [ *-{t,f} file* ] [ *-m files ...* ] [ *-r rlefile* ] [ *-o outputfile* ] [ *inputfile* ]

## DESCRIPTION

The program will load a specified color map into an *RLE(5)* file. The color map may be computed by *rleldmap* or loaded from a file in one of several formats. The input is read from *inputfile* or stdin if no file is given, and the result is written to *outputfile* or stdout.

The following terms are used in the description of the program and its options:

input map:

A color map already in the input RLE file.

applied map:

The color map specified by the arguments to *rleldmap*. This map will be applied to or will replace the input map to produce the output map.

output map:

Unless *-a* or *-b* is specified, this is equal to the applied map. Otherwise it will be the composition of the input and applied maps.

map composition:

If the applied map is composed *after* the input map, then the output map will be *applied map[input map]*. Composing the applied map before the input map produces an output map equal to *input map[applied map]*. The maps being composed must either have the same number of channels, or one of them must have only one channel. If an entry in the map being used as a subscript is larger than the length of the map being subscripted, the output value is equal to the subscript value. The output map will be the same length as the subscript map and will have the number of channels that is the larger of the two. If the input map is used as a subscript, it will be downshifted the correct number of bits to serve as a subscript for the applied map (since the color map in an *RLE(5)* file is always stored left justified in 16 bit words). This also applies to the applied map if it is taken from an *RLE(5)* file (*-r* option below). Note that if there is no input map, that the result of composition will be exactly the applied map.

**nchan:** The number of separate lookup tables (channels) making up the color map. This defaults to 3.

**length:** The number of entries in each channel of the color map. The default is 256.

**bits:** The size of each color map entry in bits. The default value is the log base 2 of the length.

**range:** The maximum value of a color map entry, equal to  $2^{\text{bits}} - 1$ .

The options have the meanings described below:

**-a** Compose the applied map *after* the input map.

**-b** Compose the applied map *before* the input map. Only one of *-a* or *-b* may be specified.

**-n *nchan length***

Specify the size of the applied map if it is not 3x256. The *length* should be a power of two, and will be rounded up if necessary. If applying the map *nchan* must be either 1 or equal to the number of channels in the input map. It may have any value if the input map has one channel or is not present.

**-s *bits*** Specify the size in bits of the color map entries.

Exactly one of the options **-l**, **-g**, **-{t,f}**, **-m**, or **-r**, must be specified.

**-l *factor***

Generate a linear applied map with the *n*th entry equal to  

$$\text{range} * \max(1.0, \text{factor} * (n / (\text{length} - 1)))$$

*Factor* defaults to 1.0 if not specified. Negative values of *factor* will generate a map with values equal to

$$\text{range} * \max(0.0, 1.0 - \text{factor} * (n / (\text{length} - 1)))$$

**-g *gamma***

Generate an applied map to compensate for a display with the given gamma. The *n*th entry is equal to

$$\text{range} * (n / (\text{length} - 1))^{1/\text{gamma}}$$

**-t *file*** Read color map entries from a table in a text file. The values for each channel of a particular entry follow each other in the file. Thus, for an RGB color map, the file would look like:

```
red0    green0  blue0
red1    green1  blue1
...     ...     ...
```

Line breaks in the input file are irrelevant.

**-f *file*** Reads the applied map from a text file, with all the entries for each channel following each other. Thus, the input file above would appear as

```
red0 red1 red2 ... (length values)
green0 green1 green2 ... (length values)
blue0 blue1 blue2 ... (length values)
```

As above, line breaks are irrelevant.

**-m *files ...***

Read the color map for each channel from a separate file. The number of files specified must equal the number of channels in the applied map. (Note: the list of files must be followed by another flag argument or by the null flag — to separate it from the *inputfile* specification.

**-o *outputfile***

The output will be written to the file *outputfile* if this option is specified. Otherwise the output will go to stdout.

*inputfile* The input will be taken from this file if specified. Otherwise, the input will be read from stdin.

SEE ALSO

*applymap*(1), *RLE*(5).

**AUTHOR**

Spencer W. Thomas, University of Utah

**BUGS**

None known.

## NAME

rlemandl - Compute images of the Mandlebrot set.

## SYNOPSIS

rlemandl real imaginary width [ -s xsize ysize ] [ -v ] > output

## DESCRIPTION

*Rlemandl* computes images of the Mandlebrot set as an eight bit gray scale image. The *real* and *imaginary* arguments specify the center of the area in the complex plane to be examined. *Width* specifies the width area to be examined.

The following options are available:

-v      Print a message after every 50 lines are generated.

-s xsize ysize  
Specify the resolution of the image (in pixels).

## SEE ALSO

"Computer Recreations," Scientific American, August 1985.

## AUTHOR

John W. Peterson, University of Utah.

## BUGS

What a frob. Gratuitous features are left as exercise to the reader.

**NAME**

`rlepatch` – patch smaller RLE files over a larger image.

**SYNOPSIS**

`rlepatch imgfile patchfiles... > output`

**DESCRIPTION**

*Rlepatch* puts smaller RLE files on top of a larger RLE image. One use for *rlepatch* is to place small "fix" images on top of a larger image that took a long time to compute. Along with *repos(1)*, *rlepatch* can also be used as a simple way to build image mosaics.

Unlike *comp(1)*, *rlepatch* does not perform any arithmetic on the pixels. If the patch images overlap, the patches specified last cover those specified first.

**SEE ALSO**

*COMP(1)*, *CROP(1)*, *REPOS(1)*

**AUTHOR**

John W. Peterson, University of Utah.

**BUGS**

*Rlepatch* uses the "row" interface to the RLE library. It would run much faster using the "raw" interface, particularly for placing small patches over a large image. Even fixing it to work like *comp* (which uses "raw" mode only for non-overlapping images) would make a major improvement.

**NAME**

**rlesetbg** – Set the background value in the RLE header.

**SYNOPSIS**

**rlesetbg** [ **-O** ] [ **-c** *red green blue* ] *inputfile*

**DESCRIPTION**

*rlesetbg* sets the background color field in the image's header (none of the actual pixels are changed). If *inputfile* isn't specified, the image is read from stdin. The flags are:

**-O** Specifies that the image has no background, it overlays existing images.

**-c** *red green blue*  
Specifies red, green and blue pixel values to set the background to.

**AUTHORS**

John W. Peterson and Rod Bogart

**SEE ALSO**

*REPOS(1)*

**BUGS**

This should really be part of a single program that does all header munging...

**NAME**

rlesplit – split a file of concatenated RLE images into separate image files

**SYNOPSIS**

**rlesplit** [ **-n** number [ digits ] ] [ **-p** prefix ] [ rlefile ]

**DESCRIPTION**

This program will split a file containing a concatenated sequence of *RLE(5)* images into separate files, each containing a single image. The output file names will be constructed from the input file name or a specified prefix, and a sequence number. If an input *rlefile* is specified, then the output file names will be in the form "*rlefileroot#.rle*", where *rlefileroot* is *rlefile* with any ".rle" suffix stripped off. If the option **-p prefix** is specified, then the output file names will be of the form "*prefix#.rle*". If neither option is given, then the output file names will be in the form "*#.rle*". Input will be read from *rlefile* if specified, from standard input, otherwise. File names will be printed on the standard error output as they are generated.

The option **-n** allows specification of an initial sequence number, and optionally the number of digits used for the sequence number. By default, numbering starts at 1, and numbers are printed with 3 digits (and leading zeros).

**SEE ALSO**

*RLE(5)*.

**AUTHOR**

Spencer W. Thomas

## NAME

rleswap - swap the channels in an RLE file.

## SYNOPSIS

rleswap [ -v ] [ -i input-channels,... ] [ -o output-channels,... ] [ -d delete-channels,... ] [ -p channel-pairs,... ] [ rlefile ]

## DESCRIPTION

This program can be used to select or swap the color channels in a *RLE(5)* file. The major options provide four different ways of specifying a mapping between the channels in the input file and the output file. Only one of the options *-i*, *-o*, *-d*, or *-p* may be specified. If the optional *rlefile* is not given, input will be read from standard input. A new *RLE(5)* file will be written to the standard output, similar to the input, except for the specified channel remappings.

The following options are available:

- v      Print the channel mappings that will be performed on the standard error output.
- i      Following this option is a comma separated list of numbers indicating the input channel that maps to each output channel in sequence. I.e., the first number indicates the input channel mapping to output channel 0. The alpha channel will be passed through unchanged if present. Any input channels not mentioned in the list will not appear in the output.
- o      Following this option is a comma separated list of numbers indicating the output channel to which each input channel, in sequence, will map. I.e., the first number gives the output channel to which the first input channel will map. No number may be repeated in this list. The alpha channel will be passed through unchanged if present. Any output channel not mentioned in the list will not receive image data. If there are fewer numbers in the list than there are input channels, the excess input channels will be ignored. If there are more numbers than input channels, it is an error.
- d      Following this option is a comma separated list of numbers indicating channels to be deleted from the input file. All other channels will be passed through unchanged.
- p      Following this option is a comma separated list of pairs of channel numbers. The first channel of each pair indicates a channel in the input file that will be mapped to the the channel in the output file indicated by the second number in the pair. No output channel number may appear more than once. Any input channel not mentioned will not appear in the output file. Any output channel not mentioned will not receive image data.

## SEE ALSO

*RLE(5)*.

## AUTHOR

Spencer W. Thomas, University of Utah



**NAME**

*rletopaint* – convert an RLE file to MacPaint format using dithering

**SYNOPSIS**

*rletopaint* [ -l ] [ -i ] [ -g [ gamma ] ] [ infile ] > outfile.rle

**DESCRIPTION**

*rletopaint* converts a file from RLE format to MacPaint format. The program uses dithering to convert from a full 24 bit color image to a bitmapped image. If the RLE file is larger than a MacPaint image (576×720) it is cropped to fit.

Because MacPaint files have their coordinate origin in the upper left instead of the lower left, the RLE file should be piped through *rleflip*(1) -v before *rletopaint*.

The following options are available:

- l      Use a linear map in the conversion from 24 bits to bitmapped output.
- g [ gamma ]  
        Use a gamma map of gamma (gamma is 2.0 if not specified)
- i      Invert the sense of the output pixels (black on white vs. white on black).

**SEE ALSO**

*painttorle*(1). *RLE*(5).

**AUTHOR**

John W. Peterson. Byte compression routine by Jim Schimpf.

**BUGS**

Should use a color map in the file, if present.

## NAME

*rletops* - Convert RLE images to PostScript

## SYNOPSIS

*rletops* [ *-h height* ] [ *-s* ] [ *-c center* ] [ *-a aspect* ] [ *infile* ]

## DESCRIPTION

*rletops* converts RLE images into PostScript. The conversion uses the PostScript *image* operator, instructing the device to reproduce the image to the best of its abilities. If *infile* isn't specified, the RLE image is read from stdin.

The following options are available:

*-h height*

Specifies the height (in inches) the image is to appear on the page. The default is three inches. The width of the image is calculated from the image height, aspect ratio, and pixel dimensions.

*-s*

Specifies image is to be generated in "Scribe Mode." The image is generated without a PostScript *showpage* operator at the end, and the default image center is changed to 3.25 inches from the margin (which usually is 1 inch). This is to generate PostScript files that can be included in Scribe documents with the *@Picture* command. Images may also be included in LaTeX documents with local conventions like the *\special{psfile=image.ps}* command.

*-c center*

Centers the images about a point *center* inches from the left edge of the page (or left margin if *-s* is specified). Default is 4.25 inches.

*-a aspect*

Specify aspect ratio of image. Default is 1.0 (note PostScript uses square pixels).

## NOTES

On devices like the Apple LaserWriter, *rletops* generates large PostScript files that take a non-trivial amount of time to download and print. A 512x512 image takes about ten minutes. For including images in documents at the default sizes, 256x256 is usually sufficient resolution.

## AUTHORS

Rod Bogart and John W. Peterson.

Portions are based on a program by Marc Majka.

## BUGS

Due to a mis-understanding with the PostScript interpreter, *rletops* always rounds the image size up to an even number of scanlines.

**NAME**

*rlezoom* - Magnify an RLE file by an integral factor.

**SYNOPSIS**

*rlezoom* factor [ y-factor ] [ rlefile ]

**DESCRIPTION**

This program magnifies (zooms) an *RLE(5)* file by an integral factor. Each pixel in the original image becomes a block of pixels in the output image. If no *y-factor* is specified, then the image will be magnified by *factor* equally in both directions. If *y-factor* is given, then each input pixel becomes a block of *factor*  $\times$  *y-factor* pixels in the output. Input is taken from *rlefile*, or from the standard input if not specified. The magnified image is written to the standard output.

You should use *rlezoom* over *fant(1)* if you just want an integer magnification of an image with the pixel boundaries showing. It is significantly faster than *fant* in this case. If you need blending between pixels in the magnified image, then *fant* is the correct program to use.

**SEE ALSO**

*fant(1)*. *RLE(5)*.

**AUTHOR**

Spencer W. Thomas

**NAME**

*smush* - defocus an RLE image.

**SYNOPSIS**

*smush* [ levels ] [ -m maskfile ] [ -n ] [ -o outfile ] [ infile ]

**DESCRIPTION**

*smush* convolves an image with a 5x5 Gaussian mask, blurring the image. One may also provide a mask in a text file. The file must contain an integer to specify the size of the square mask, followed by size\*size floats. The mask will be normalized (forced to sum to 1.0) unless the -n flag is given.

The resulting image is the same size as the input image, no sub-sampling takes place. The levels option, which defaults to one, signifies the number of times which the image will be blurred. Each successive blurring is done with a more spread out mask, so a *smush* of level 2 is blurrier than piping two level one *smush* calls. If no input file is specified, *smush* reads from stdin. If no output file is specified with -o it writes the result to stdout.

**SEE ALSO**

*avg4*(1)

**AUTHOR**

Rod G. Bogart

**BUGS**

*smush* should probably automatically generate different sized gaussians and other common filters.

**NAME**

**to8** - Convert a 24 bit RLE file to eight bits using dithering.

**SYNOPSIS**

**to8** [ **-(iI)** *image\_gamma* ] [ **-g** *display\_gamma* ] [ **-o** *outfile* ] [ *infile* ]

**DESCRIPTION**

**to8** Converts an image with 24 bit pixel values (eight bits of red, green and blue) to eight bits of color using a dithered color map (the special color map is automatically written into the output file). If no input file is specified, *tobw* reads from stdin. If no output file is specified with **-o** it writes the result to stdout.

Other options allow control over the gamma, or contrast, of the image. The dithering process assumes that the incoming image has a gamma of 1.0 (i.e., a 200 in the input represents an intensity twice that of a 100.) If this is not the case, the input values must be adjusted before dithering via the **-i** or **-I** option. The input file may also specify the gamma of the image via a picture comment (see below). The output display is assumed to have a gamma of 2.5 (standard for color TV monitors). This may be modified via the **-g** option if a display with a different gamma is used.

**to8** will put a picture comment into the output file indicating the display gamma assumed in constructing the dithering color map.

The following options are available:

**-i image\_gamma**

Specify the gamma (contrast) of the image. A low contrast image, suited for direct display without compensation on a high contrast monitor (as most monitors are) will have a gamma of less than one. The default image gamma is 1.0. Image gamma may also be specified by a picture comment in the *RLE (5)* file of the form **image\_gamma=gamma**. The command line argument will override the value in the file if specified.

**-I image\_gamma**

An alternate method of specifying the image gamma, the number following **-I** is the gamma of the display for which the image was originally computed (and is therefore 1.0 divided by the actual gamma of the image). Image display gamma may also be specified by a picture comment in the *RLE (5)* file of the form **display\_gamma=gamma**. The command line argument will override the value in the file if specified.

**-g display\_gamma**

Specify the gamma of the *X* display monitor. The default value is 2.5, suitable for most color TV monitors (this is the gamma value assumed by the NTSC video standard).

**SEE ALSO**

*tobw*(1), *getX*(1),

**AUTHOR**

Spencer Thomas

**NAME**

*tobw* - Convert a 24 bit RLE file to eight bits of gray scale value.

**SYNOPSIS**

*tobw* [ -t ] [ -o outfile ] [ infile ]

**DESCRIPTION**

*tobw* Converts an image with 24 bit pixel values (eight bits of red, green and blue) to eight bits of grayscale information. If the -t flag is given, then the monochrome pixel values are replicated on all three output channels (otherwise, just one channel of eight bit data is produced). If no input file is specified, *tobw* reads from stdin. If no output file is specified with -o, it writes the result to stdout.

**SEE ALSO**

*to8*(1).

**AUTHOR**

Spencer Thomas

**NAME**

**unexp** - Convert "exponential" files into normal files.

**SYNOPSIS**

**unexp** [ **-p** ] [ **-v** ] [ **-s** ] [ **-m maxval** ] infile [ outfile ]

**DESCRIPTION**

*Unexp* Converts a file of "exponential" floating point values into an rle file containing integer valued bytes. Exponential files have N-1 channels of eight bit data, with the Nth channel containing a common exponent for the other channels. This allows the values represented by the pixels to have a wider dynamic range.

If no maximum value is specified, *unexp* first reads the RLE file to find the dynamic range of the whole file. It then rewinds the file and scales the output to fit within that dynamic range. If a maximum value is specified, *unexp* runs in one pass, and clamps any values exceeding the maximum.

Files containing exponential data are expected to have a "exponential\_data" comment; *unexp* prints a warning if such a comment doesn't exist. An exponential file should be *unexp*'ed before attempting to use any tools that perform arithmetic on pixels (e.g., *comp*, *avg4*, *fant*, *rleldmap*, etc) or displaying the image.

The following options are available:

- p**     Print the maximum value found during the scanning phase
- v**     Verbose mode, print a message to stderr after scanning or converting every hundred scanlines.
- m maxval**  
       Specify the maximum value (i.e., the data in the file is assumed to be in the range 0..maxval).  
       Only the conversion pass is executed, and values found exceeding the maximum are clamped.
- s**     Just scan the file to find the maximum, don't generate any output.

**SEE ALSO**

*float\_to\_exp*(3), *RLE*(5).

**AUTHOR**

John W. Peterson

**BUGS**

*Unexp* is provided for the lack of floating point or extended precision RLE files.

The **-v** flag is a historical relic.

## NAME

**unslice** - Quickly assemble image slices

## SYNOPSIS

**unslice** [ **-f** *ctlfile* ] [ **-y** *ymax* ] infile ... > outfile

## DESCRIPTION

*Unslice* quickly assembles a number of horizontal image strips into a single output image. A typical use for *unslice* is to put together portions of an image ("slices") computed independantly into a single output picture. Because *unslice* uses the "raw" RLE library calls to read and write the images, it runs much faster than doing the equivalent operations with *crop* and *comp*.

*Unslice* has two modes of operation. If given the **-f** flag, *unslice* reads a control file telling it how to assemble the images. This is a text file with two decimal numbers on each line, one line for each slice to be assembled into the output image. Each line gives the starting and stopping scanlines (inclusive) for each slice. These must be in ascending order. This is useful if the slices have excess image area that should be cropped away.

If no control file is given, the **-y** flag is used. This tells *unslice* what the maximum Y value of the output image is. *Unslice* reads the files in order, using the RLE headers to determine where to place the slices. If two slices overlap the first scanlines from the second slice are thrown away. In both cases, the slices must be in ascending order, and are expected to be of uniform width.

## SEE ALSO

*crop(1)*, *comp(1)*, *repos(1)*.

## AUTHOR

John W. Peterson





# **Toolkit Programmer's Guide**



## NAME

rle – Run length encoded file format produced by svfb

## DESCRIPTION

The output file format is (note: all words are 16 bits, and in PDP-11 byte order):

Word 0 A "magic" number 0x4c52. (Byte order 0x52, 0x4c.)

## Words 1-4

The structure (chars saved in PDP-11 order)

```
{
    short  xpos,          /* Lower left corner
        ypos,
        xsize,          /* Size of saved box
        ysize;
}
```

Byte 10 (*flags*) The following flags are defined:

*H\_CLEARFIRST*

(0x1) If set, clear the frame buffer to background color before restoring.

*H\_NO\_BACKGROUND*

(0x2) If set, no background color is supplied. If *H\_CLEARFIRST* is also set, it should be ignored (or alternatively, a clear-to-black operation could be performed).

*H\_ALPHA*

(0x4) If set, an alpha channel is saved as color channel -1. The alpha channel does not contribute to the count of colors in *ncolors*.

*H\_COMMENT*

(0x8) If set, comments will follow the color map in the header.

Byte 11 (*ncolors*) Number of color channels present. 0 means load only the color map (if present), 1 means a B&W image, 3 means a normal color image.

Byte 12 (*pixelbits*) Number of bits per pixel, per color channel. Values greater than 8 currently will not work.

Byte 13 (*ncmap*) Number of color map channels present. Need not be identical to *ncolors*. If this is non-zero, the color map follows immediately after the background colors.

Byte 14 (*cmaplen*) Log base 2 of the number of entries in the color map for each color channel. I.e., would be 8 for a color map with 256 entries.

## Bytes 15-...

The background color. There are *ncolors* bytes of background color. If *ncolors* is even, an extra padding byte is inserted to end on a 16 bit boundary. The background color is only present if *H\_NO\_BACKGROUND* is not set in *flags*. If *H\_NO\_BACKGROUND* is set, there is a single filler byte. Background color is ignored, but present, if *H\_CLEARFIRST* is not set in *flags*.

If *ncmap* is non-zero, then the color map will follow as *ncmap*\*2<sup>*cmaplen*</sup> 16 bit words. The color map data is left justified in each word.

If the *H\_COMMENT* flag is set, a set of comments will follow. The first 16 bit word gives the length of the comments in bytes. If this is odd, a filler byte will be appended to the comments. The comments are interpreted as a sequence of null terminated strings which should be, by convention, of the form *key=value*.

Following the setup information is the Run Length Encoded image. Each instruction consists of an opcode, a datum and possibly one or more following words (all words are 16 bits). The opcode is encoded in the first byte of the instruction word. Instructions come in either a short or long form. In the short form, the datum is in the second byte of the instruction word; in the long form, the datum is a 16 bit value in the word following the instruction word. Long form instructions are distinguished by having the 0x40 bit set in the opcode byte. The instruction opcodes are:

**SkipLines (1)**

The datum is an unsigned number to be added to the current Y position.

**SetColor (2)**

The datum indicates which color is to be loaded with the data described by the following ByteData and RunData instructions. Typically, 0→red, 1→green, 2→blue. The operation also resets the X position to the initial X (i.e. a carriage return operation is performed).

**SkipPixels (3)**

The datum is an unsigned number to be added to the current X position.

**ByteData (5)**

The datum is one less than the number of bytes of color data following. If the number of bytes is odd, a filler byte will be appended to the end of the byte string to make an integral number of 16-bit words. The X position is incremented to follow the last byte of data.

**RunData (6)**

The datum is one less than the run length. The following word contains (in its lower 8 bits) the color of the run. The X position is incremented to follow the last byte in the run.

**EOF (7)**

This opcode indicates the logical end of image data. A physical end-of-file will also serve as well. The EOF opcode may be used to concatenate several images in a single file.

**AUTHOR**

Spencer W. Thomas, Todd Fuqua

**SEE ALSO**

*svfb(3)*

## NAME

**buildmap** – create a color map array from an RLE file header.

## SYNOPSIS

```
#include <svfb_global.h>

rle_pixel ** buildmap( globals, minmap, gamma )
struct sv_globals * globals;
double gamma;
```

## DESCRIPTION

The color map in the *sv\_globals*(3) structure is not in the most easily used form. The function *buildmap* returns a pointer to a colormap array with certain minimum dimensions, making it easy to implement color mapping in a program. The color map from first argument, *globals*, is used to build the result. If no map is present in *globals*, then an identity map of the minimum size will be returned.

The returned color map will have at least *minmap* rows or channels, each of which is at least 256 entries long (so that indexing into the color map with an 8 bit *rle\_pixel* value will always succeed.)

The color map from *globals* will be composed with a gamma compensation curve to account for the gamma of the display for which the color map was presumably computed. The argument *gamma* specifies the gamma of the compensation curve. It would typically be the reciprocal of the display gamma.

The returned value is a pointer to an array of pointers to arrays of *rle\_pixel* values. It may be doubly indexed in C code, so that if *cmap* is the return value, the RGB color mapping for a pixel *pixval* is (*cmap*[0][*pixval*], *cmap*[1][*pixval*], *cmap*[2][*pixval*]).

## AUTHOR

Spencer W. Thomas  
University of Utah

## FILES

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

## SEE ALSO

*RLE*(5), *sv\_globals*(3).

## NAME

*dither*, *dithermap*, *bwdithermap*, *make\_square*, *dithergb*, *ditherbw* – functions for dithering color or black and white images.

## SYNOPSIS

```
dithermap( levels, gamma, rgbmap, divN, modN, magic )
double gamma;
int rgbmap[][3];
int divN[256];
int modN[256];
int magic[16][16];

bwdithermap( levels, gamma, bwmap, divN, modN, magic )
double gamma;
int bwmap[];
int divN[256];
int modN[256];
int magic[16][16];

make_square( N, divN, modN, magic )
double N;
int divN[256];
int modN[256];
int magic[16][16];

dithergb( x, y, r, g, b, levels, divN, modN, magic )
int divN[256];
int modN[256];
int magic[16][16];

ditherbw( x, y, val, divN, modN, magic )
int divN[256];
int modN[256];
int magic[16][16];
```

## DESCRIPTION

These functions provide a common set of routines for dithering a full color or gray scale image into a lower resolution color map.

*dithermap* computes a color map and some auxiliary parameters for dithering a full color (24 bit) image to fewer bits. The argument *levels* tells how many different intensity levels per primary color should be computed. To get maximum use of a 256 entry color map, use *levels*=6. The *gamma* argument provides for gamma compensation of the generated color map. The computed color map will be returned in the array *rgbmap*. *divN* and *modN* are auxiliary arrays for computing the dithering pattern (see below), and *magic* is the magic square dither pattern.

To compute a color map for dithering a black and white image to fewer intensity levels, use *bwdithermap*. The arguments are as for *dithermap*, but only a single channel color map is computed.

To just build the magic square and other parameters, use *make\_square*. The argument *N* should be equal to 255.0 divided by the desired number of intensity levels. The other arguments are filled in as above.

The color map index for a dithered full color pixel is computed by *dithergb*. Since the pattern depends on the screen location, the first two arguments *x* and *y*, specify that location. The true color of the pixel at that location is given by the triple *r*, *g*, and *b*. The number of intensity *levels* and the dithering parameter

matrices computed by *dithermap* are also passed to *dithergb*.

The color map index for a dithered gray scale pixel is computed by *ditherbw*. Again, the screen position is specified, and the intensity value of the pixel is supplied in *val*. The dithering parameters must also be supplied.

Alternatively, the dithering may be done in line instead of incurring the extra overhead of a function call, which can be significant when repeated a million times. The computation is as follows:

```
row = y % 16;
col = x % 16;
#define DMAP(v,col,row) (divN[v] + (modN[v]>magic[col][row] ? 1 : 0))
pix = DMAP(r,col,row) + DMAP(g,col,row)*levels +
      DMAP(b,col,row)*levels*levels;
```

For a gray scale image, it is a little simpler:

```
pix = DMAP(row,col,val);
```

And on a single bit display:

```
pix = divN[val] > magic[col][row] ? 1 : 0
```

#### AUTHOR

Spencer W. Thomas  
University of Utah

#### FILES

On *utah-gr* the library is available as *-lrle*.

#### SEE ALSO



## NAME

`float_to_exp` – Convert floating point values into "exponential" pixels.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
rle_row_alloc( count, floats, pixels )  
int count;  
float * floats;  
rle_pixel * pixels;
```

## DESCRIPTION

The function *float\_to\_exp* converts *count* floating point numbers (point to by *floats*) into *count+1* bytes (pointed to by *pixels*) using an "exponential" format. This format generates *count* pixels as eight bit "mantissa" values, and another byte containing a common exponent for all of the data values. This format has a wider dynamic range of values with little extra overhead.

Files containing exponential data may be converted into displayable images using the *unexp*(1) command. *Unexp* should be used before using any tools that perform arithmetic on pixel values, or displaying the image.

*Unexp* expects files containing exponential data to have an "exponential\_data" picture comment.

## AUTHOR

John W. Peterson, based on code by Spencer Thomas.  
University of Utah

## FILES

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

## SEE ALSO

*RLE*(5), *unexp*(1),

## NAME

rgb\_to\_bw – convert a color scanline to black and white.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
void rgb_to_bw( red_row, green_row, blue_row, bw_row, length );  
rle_pixel * red_row, * green_row, * blue_row, *bw_row;  
int length;
```

## DESCRIPTION

*rgb\_to\_bw* converts red/green/blue color information to black and white using the *NTSC Y* transform:  
 $Y = 0.35 * R + 0.55 * G + 0.10 * B$ . The arguments point to scanlines with *length* bytes in each. *bw\_row* may be identical to one of *red\_row*, *green\_row*, or *blue\_row*.

## AUTHOR

Spencer W. Thomas  
University of Utah

## FILES

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

## SEE ALSO

**NAME**

`rle_delcom` – delete a picture comment from an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
char * rle_delcom( name, globals )
```

```
char * name;
```

```
struct sv_globals * globals;
```

**DESCRIPTION**

`rle_delcom` is used to delete a picture comment from a `sv_globals` structure. It is called with the *name* of the comment and the *globals* structure to be modified. The first comment in the `sv_globals` structure of the form *name=value* or *name.* will be deleted. The deleted comment will be returned as the function value.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On `utah-gr` the include file is in `/usr/site/include` and the library is available as `-lrle`.

**SEE ALSO**

`RLE(5)`, `sv_globals(3)`, `rle_getcom(3)`, `rle_putcom(3)`, `svfb(3)`.

**NAME**

*rle\_freeraw* – Free pixel storage allocated by *rle\_getraw*.

**SYNOPSIS**

```
#include <svfb_global.h>
#include <rle_getraw.h>
```

```
rle_freeraw( globals, scanraw, nraw );
struct sv_globals * globals;
rle_op ** scanraw;
int * nraw;
```

**DESCRIPTION**

The pixel storage allocated dynamically by *rle\_getraw*(3) must be freed to avoid memory leaks. This is most easily accomplished by calling *rle\_freeraw*. The argument *scanraw* points to an array of *rle\_op* structures, with *nraw* indicating the number of structures in each channel. All pixel data arrays will be freed by the call to *rle\_freeraw*.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

**SEE ALSO**

*RLE*(5), *sv\_globals*(3), *rle\_raw\_alloc*(3), *rle\_raw\_free*(3), *rle\_getraw*(3), *svfb*(3).

**NAME**

*rle\_get\_setup* – Read the header from an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>

rle_get_setup( globals );
struct sv_globals * globals;

rle_get_setup_ok( globals, prog_name, file_name );
struct sv_globals * globals;
char * prog_name;
char * file_name;
```

**DESCRIPTION**

*rle\_get\_setup* is called to initialize the process of reading an RLE file. It will fill in the *globals* structure with the header information from the RLE file, and will initialize state for *rle\_getrow*. The *svfb\_fd* field of the *globals* structure should be initialized to the input stream before calling *rle\_get\_setup*. The *sv\_bits* field is initialized by *rle\_get\_setup* to enable reading of all the channels present in the input file. To prevent *rle\_getrow* (3) from reading certain channels (e.g., the alpha channel), the appropriate bits should be cleared before calling *rle\_getrow*(3).

*rle\_get\_setup\_ok* invokes *rle\_getrow* and checks the results. If an error occurs, the appropriate error message is printed on stderr, and the program exits with the status code set. The *prog\_name* and *file\_name* parameters are used for the error message. If *file\_name* is NULL, the string "stdin" is substituted.

**AUTHOR**

Spencer W. Thomas, Todd Fuqua  
University of Utah

**FILES**

On utah-gr the include file is in /usr/site/include and the library is available as -lrle.

**SEE ALSO**

*RLE*(5), *sv\_globals*(3), *rle\_getrow*(3), *rle\_getraw*(3), *svfb*(3).

**NAME**

`rle_getcom` – get a picture comment from an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
char * rle_getcom( name, globals )  
char * name;  
struct sv_globals * globals;
```

**DESCRIPTION**

`rle_getcom` returns a pointer to the data portion of a picture comment from an RLE file. The comment is assumed to be in the form *name=value*; a pointer to *value* is returned. If the comment is of the form *name*, a pointer to the null character at the end of the string is returned. If there is no comment of the above forms, a *NULL* pointer is returned. The *globals* structure contains the picture comments in question.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On `utah-gr` the include file is in `/usr/site/include` and the library is available as `-lrle`.

**SEE ALSO**

`RLE(5)`, `sv_globals(3)`, `rle_putcom(3)`, `rle_delcom(3)`, `svfb(3)`.

## NAME

*rle\_getraw* - Read run length encoded information from an RLE file.

## SYNOPSIS

```
#include <svfb_global.h>
#include <rle_getraw.h>

rle_getraw( globals, scanraw, nraw );
struct sv_globals * globals;
rle_op ** scanraw;
int * nraw;
```

## DESCRIPTION

*rle\_getraw* can be used to read information from an RLE file in the "raw" form. The arguments *scanraw* and *nraw* are identical to those for *sv\_putraw*(3), except that the information will be filled in by *rle\_getraw*. However, sufficient space must be allocated in the arrays of *rle\_op* structures to hold the data read from the file. A function *rle\_raw\_alloc*(3) (q.v.) is provided to make this easier. A file created with *sv\_putrow* will require at most  $(sv\_globals \rightarrow sv\_xmax - sv\_globals \rightarrow sv\_xmin) / 3$  elements per channel. The storage required by any pixel sequences in the input will be dynamically allocated by *rle\_getraw*. To free this space, call *rle\_freeraw*(3).

## AUTHOR

Spencer W. Thomas  
University of Utah

## FILES

On utah-gr the include file is in /usr/site/include and the library is available as -lrle.

## SEE ALSO

*RLE*(5), *sv\_globals*(3), *sv\_putraw*(3), *rle\_raw\_alloc*(3), *rle\_raw\_free*(3), *rle\_getrow*(3), *rle\_freeraw*(3), *svfb*(3).

## NAME

`rle_getrow` – Read a scanline of pixels from an RLE file.

## SYNOPSIS

```
#include <svfb_global.h>

rle_getrow( globals, rows );
struct sv_globals * globals;
rle_pixel ** rows;
```

## DESCRIPTION

`rle_getrow` reads information for a single scanline from the input file each time it is called. *globals* should point to the structure initialized by `rle_get_setup`. The array *rows* should contain pointers to arrays of characters, into which the scanline data will be written. There should be as many elements in *rows* as there are primary colors in the input file (typically 1 or 3), and the scanline arrays must be indexable up to the maximum X coordinate, as specified by *globals*→*sv\_xmax*. `rle_getrow` returns the y value of the scanline just read. This will always be 1 greater than the y value from the scanline previously read, and starts at *globals*→*sv\_ymin*. Only those channels enabled by *globals*→*sv\_bits* will be returned.

Note: `rle_getrow` will continue to return scanlines even after the end of the input file has been reached, incrementing the return scanline number each time it is called. The calling program should use some other termination criterion (such as the scanline number reaching *globals*→*sv\_ymax*, or explicitly testing for end of file on the input with `feof(infile)`). Thesecondtestmay `rle_getrow` has encountered a logical EOF in the file. The first will always work eventually.)

For example, the code below reads the first two 3 color scanlines of 512 pixels from an RLE file on the standard input.

```
char scanline[2][3][512], *rows[3];
int row, i;
sv_globals.svfb_fd = stdin;
rle_get_setup( &sv_globals );
for ( row = 0; row < 2; row++ )
{
    for ( i = 0; i < 3; i++ )
        rows[i] = scanline[row][i];
    rle_getrow( &sv_globals, rows );
}
```

## AUTHOR

Spencer W. Thomas, Todd Fuqua  
University of Utah

## FILES

On `utah-gr` the include file is in `/usr/site/include` and the library is available as `-lrle`.

## SEE ALSO

`RLE(5)`, `sv_globals(3)`, `rle_row_alloc(3)`, `rle_row_free(3)`, `rle_get_setup(3)`, `rle_getraw(3)`, `svfb(3)`.



**NAME**

**rle\_putcom** – set the value of a picture comment in an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
char * rle_putcom( value, globals )  
char * value;  
struct sv_globals * globals;
```

**DESCRIPTION**

*rle\_putcom* can be used to add a picture comment or change the value of a picture comment in a *sv\_globals(3)* structure. The argument *value* is the string value of the comment, and is generally of the form *name=value*. It may also be of the form *name*. If there is another comment with the same *name*, it will be replaced with the new *value*, and the previous comment will be returned as the value of *rle\_putcom*.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

**SEE ALSO**

*RLE(5)*, *sv\_globals(3)*, *rle\_getcom(3)*, *rle\_delcom(3)*, *svfb(3)*.

## NAME

*rle\_raw\_alloc* - Allocate memory for *rle\_getraw* or *sv\_putraw*.

## SYNOPSIS

```
#include <svfb_global.h>
#include <rle_getraw.h>
```

```
rle_raw_alloc( globals, scanp, nrawp )
struct sv_globals * globals;
rle_op *** scanp;
int ** nrawp;
```

## DESCRIPTION

The function *rle\_raw\_alloc* is provided to make it easier to allocate storage for use by the RLE "raw" functions. They examine the *globals* structure provided and return (via their other arguments) newly allocated space suitable for reading from or writing to an RLE file described by the *globals* structure. It allocates a maximum amount of storage for each channel. *rle\_raw\_alloc* allocates *globals*→*sv\_xmax* - *globals*→*sv\_xmin* elements per channel. To free this storage, call *rle\_raw\_free*(3). This is distinct from *rle\_freeraw*(3), since it frees all the storage consumed by the arrays of pointers and *rle\_op* structures, while only frees pixel arrays referenced by individual *rle\_op* structures. In fact, *rle\_freeraw*(3) should be called before calling *rle\_raw\_free*.

## AUTHOR

Spencer W. Thomas  
University of Utah

## FILES

On utah-gr the include file is in /usr/site/include and the library is available as -lrle.

## SEE ALSO

*RLE*(5), *sv\_globals*(3), *sv\_setup*(3), *sv\_putraw*(3), *rle\_raw\_free*(3), *rle\_get\_setup*(3), *rle\_getraw*(3), *rle\_freeraw*(3), *svfb*(3).

## NAME

`rle_raw_free` – free memory allocated by `rle_raw_alloc`.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
#include <rle_getraw.h>
```

```
rle_raw_free( globals, scanp, nrawp ) struct sv_globals * globals;
```

```
rle_op ** scanp;
```

```
int * nrawp;
```

## DESCRIPTION

`rle_raw_free` should be used to free memory allocated by `rle_raw_alloc(3)`. The arguments are pointers to the allocated storage. This is distinct from `rle_freeraw(3)`, since it frees all the storage consumed by the arrays of pointers and `rle_op` structures, while only frees pixel arrays referenced by individual `rle_op` structures. In fact, `rle_freeraw(3)` should be called before calling `rle_raw_free`.

## NAME

*rle\_row\_alloc* – Allocate scanline memory for *sv\_putrow* or *rle\_getrow*.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
rle_row_alloc( globals, scanp )  
struct sv_globals * globals;  
rle_pixel *** scanp;
```

## DESCRIPTION

The function *rle\_row\_alloc* is provided to make it easier to allocate storage for use by the RLE functions. It examines the *globals* structure provided and returns (via its other argument) newly allocated space suitable for reading from or writing to an RLE file described by the *globals* structure. *rle\_row\_alloc* allocates *globals*→*sv\_xmax* bytes for each scanline, to allow for *rle\_getrow* usage. To free this storage, call *rle\_row\_free*.

## AUTHOR

Spencer W. Thomas  
University of Utah

## FILES

On *utah-gr* the include file is in */usr/site/include* and the library is available as *-lrle*.

## SEE ALSO

*RLE*(5), *sv\_globals*(3), *sv\_setup*(3), *sv\_putrow*(3), *rle\_row\_free*(3), *rle\_get\_setup*(3), *rle\_getrow*(3), *svfb*(3).

**NAME**

*rle\_row\_free* – Free scanline memory allocated by *rle\_row\_alloc*.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
rle_row_free( globals, scanp )  
struct sv_globals * globals;  
rle_pixel ** scanp;
```

**DESCRIPTION**

To free memory allocated by *rle\_row\_alloc*(3), call *rle\_row\_free* with the pointer to the allocated storage.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On utah-gr the include file is in /usr/site/include and the library is available as *-lrle*.

**SEE ALSO**

*RLE*(5), *sv\_globals*(3), *rle\_row\_alloc*(3), *svfb*(3).

**NAME**

svfb – Functions to create and read Run Length Encoded image files.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
cc ... -lrle
```

**DESCRIPTION**

The *RLE(5)* image file format provides a method for saving and restoring images in a device independent form. A number of subroutines are available to facilitate write and reading *RLE(5)* files. They are described separately in their own manual pages (included below).

**AUTHOR**

Spencer W. Thomas, Todd Fuqua  
University of Utah

**FILES**

On utah-gr the include file is in /usr/site/include and the library is available as *-lrle*.

**SEE ALSO**

*RLE(5)*, *sv\_globals(3)*, *sv\_setup(3)*, *sv\_putrow(3)*, *sv\_putraw(3)*, *sv\_puteof(3)*, *rle\_row\_alloc(3)*, *rle\_raw\_alloc(3)*, *rle\_row\_free(3)*, *rle\_raw\_free(3)*, *rle\_get\_setup(3)*, *rle\_getrow(3)*, *rle\_getraw(3)*, *rle\_freeraw(3)*, *rle\_getcom(3)*, *rle\_putcom(3)*, *buildmap(3)*, *dither(3)*.

## NAME

sv\_globals – Structure for communication with RLE functions.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
SV_SET_BIT(globals,bit)
SV_CLR_BIT(globals,bit)
SV_BIT(globals,bit)
struct sv_globals globals;
```

## DESCRIPTION

This data structure provides communication to and between all the *RLE(5)* file routines. It describes the parameters of the image being saved or read, and contains some variables describing file state that are private to the routines.

```
struct sv_globals {
    enum sv_dispatch sv_dispatch;

    int      sv_ncolors,      /* Which set of fns to save with */
            *sv_bg_color,    /* Number of colors being saved */
            sv_alpha,        /* Background color array */
            sv_background,   /* if ≠ 0, save alpha channel (color -1) */
                                /* alpha channel background is always 0 */
                                /* if = 0, no background processing */
                                /* if = 1 or 2, save only non-bg pixels */
                                /* If 2, set clear-to-bg flag in file */
            sv_xmin,         /* Min X bound of saved raster */
            sv_xmax,         /* Max X bound */
            sv_ymin,         /* Min Y bound */
            sv_ymax,         /* Max Y bound */
            sv_ncmap,        /* number of color channels in color map */
                                /* if = 0, color map is not saved */
            sv_cmaplen;      /* Log2 of the number of entries in */
                                /* each channel of the color map */
    rle_map  *               sv_cmap; /* pointer to color map, stored as 16 bit */
                                /* words, with values left justified */
    FILE *   svfb_fd;       /* I/O to this file */
    /*
     * Bit map of channels to read/save. Indexed by (channel mod 256).
     */
    char     sv_bits[256/8];
};
```

A global variable, *sv\_globals*, is available, conveniently initialized with default values. The structure also contains some private storage used by the RLE functions.

The macro *SV\_BIT* will retrieve the state of one of the bits in the *sv\_bits* map. *SV\_SET\_BIT*, and *SV\_CLR\_BIT* set and clear bits in the *sv\_bits* map. The predefined symbols *SV\_RED*, *SV\_GREEN*, *SV\_BLUE*, and *SV\_ALPHA* may be used in these macros.

## AUTHOR

Spencer W. Thomas, Todd Fuqua

**FILES**

On utah-gr the include file is in /usr/site/include.

**SEE ALSO**

*RLE(5), sv\_globals(3), sv\_setup(3), sv\_putrow(3), sv\_putraw(3), rle\_row\_alloc(3), rle\_raw\_alloc(3), rle\_row\_free(3), rle\_raw\_free(3), rle\_get\_setup(3), rle\_getrow(3), rle\_getraw(3), rle\_freeraw(3), svfb(3).*



**NAME**

**sv\_puteof** – write a logical end of file to an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>

sv_puteof( globals );
struct sv_globals * globals;
```

**DESCRIPTION**

Call *sv\_puteof* to write a logical end of file opcode into an *RLE(5)* file.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On *utah-gr* the include file is in */usr/site/include*, and the library is *-lrle*.

**SEE ALSO**

*RLE(5)*, *sv\_globals(3)*, *sv\_setup(3)*, *sv\_putrow(3)*, *sv\_putraw(3)*, *svfb(3)*.

**NAME**

*sv\_putraw* – write run length encoded data to an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
#include <rle_getraw.h>

sv_putraw( scanraw, nraw, globals );
rle_op ** scanraw;
int * nraw;
struct sv_globals * globals;
```

**DESCRIPTION**

The function *sv\_putraw* provides a more structured method for creating run length encoded output. It is passed an array, *scanraw*, of pointers to arrays of *rle\_op* structures, and an array of lengths. Each *rle\_op* structure specifies a run or sequence of pixel values. The array *nraw* gives the number of *rle\_op* structures for each channel. I.e., *nraw[i]* is the length of the array pointed to by *scanraw[i]*.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On *utah-gr* the include file is in */usr/site/include*, and the library is *-lrle*.

**SEE ALSO**

*RLE(5)*, *sv\_globals(3)*, *sv\_setup(3)*, *sv\_putraw(3)*, *sv\_puteof(3)*, *rle\_raw\_alloc(3)*, *rle\_raw\_free(3)*, *rle\_getraw(3)*, *rle\_freeraw(3)*, *svfb(3)*.

**NAME**

**sv\_putrow** – Write a row (scanline) of data to an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
void sv_putrow( rows, length, globals );  
rle_pixel ** rows;  
int length;  
struct sv_globals * globals;
```

**DESCRIPTION**

*sv\_putrow* is called for each output scanline. *rows* is an array of pointers to the pixel data for the color components of the scanline. Rows should have *globals*→*sv\_ncolors* elements. If an alpha channel is being saved, *rows*[-1] should point to the alpha channel data. *length* is the number of pixels in the scanline.

**AUTHOR**

Spencer W. Thomas, Todd Fuqua

**FILES**

On *utah-gr*, the routines are in *-lrle*, and the include file is in */usr/site/include*.

**SEE ALSO**

*RLE*(5), *sv\_globals*(3), *sv\_putrow*(3), *sv\_skiprow*(3), *sv\_putraw*(3), *sv\_puteof*(3), *rle\_row\_alloc*(3), *rle\_raw\_alloc*(3), *rle\_row\_free*(3), *rle\_raw\_free*(3), *svfb*(3).

## NAME

`sv_setup` – setup to create an RLE file.

## SYNOPSIS

```
#include <svfb_global.h>
```

```
void sv_setup( dispatch_index, globals );  
enum { RUN_DISPATCH } dispatch_index;  
struct sv_globals * globals;
```

## DESCRIPTION

One of a group of functions for creating *RLE(5)* files.

`sv_setup` is called to initialize the output and write the image file header. The first argument specifies the type of file desired, and is, in this distribution, limited to the value **RUN\_DISPATCH**, for creating *rle(5)* files. Output functions for other file types may be created and driven from the *sv* routines. The second argument is a pointer to a *sv\_globals(3)* structure, which has been filled in with appropriate values for the image being saved.

## AUTHOR

Spencer W. Thomas, Todd Fuqua

## FILES

On *utah-gr*, the routines are in *-lrle*, and the include file is in */usr/site/include*.

## SEE ALSO

*RLE(5)*, *sv\_globals(3)*, *sv\_putrow(3)*, *sv\_putraw(3)*, *rle\_row\_alloc(3)*, *rle\_raw\_alloc(3)*, *rle\_row\_free(3)*, *rle\_raw\_free(3)*, *svfb(3)*.

**NAME**

**sv\_skiprow** – Skip output scanlines in an RLE file.

**SYNOPSIS**

```
#include <svfb_global.h>
```

```
sv_skiprow( globals, nrow )  
struct sv_globals * globals;
```

**DESCRIPTION**

This routine is used to skip blank (background) scanlines in an *RLE(5)* file. It is used in conjunction with *sv\_putrow(3)* or *sv\_putraw(3)*. The number of scanlines indicated by *nrow* will be blank in the output file.

**AUTHOR**

Spencer W. Thomas  
University of Utah

**FILES**

On utah-gr the include file is in */usr/site/include* and the source files are in */u/alpha1/urt/lib*.

**SEE ALSO**

*RLE(5)*, *sv\_globals(3)*, *sv\_setup(3)*, *sv\_putrow(3)*, *sv\_putraw(3)*, *svfb(3)*.